SANDIA REPORT

SAND2004-6252
Unlimited Release
Printed December 2004

The Common Geometry Module (CGM)

Timothy J. Tautges
Parallel Computing Sciences Department
Sandia National Laboratories

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550
Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of
Energy under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A11
Microfiche copy: A01

SAND2004-6252
Unlimited Release

Printed December 2004.

The Common Geometry Module (CGM)

Timothy J. Tautges
Parallel Computing Sciences Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-0847

Abstract

The Common Geometry Module (CGM) is a code library which provides geometry functionality used for mesh generation and other applications. This functionality includes that commonly found in solid modeling engines, like geometry creation, query and modification; CGM also includes capabilities not commonly found in solid modeling engines, like geometry decomposition tools and support for shared material interfaces. CGM is built upon the ACIS solid modeling engine, but also includes geometry capability developed beside and on top of ACIS. CGM can be used as-is to provide geometry functionality for codes needing this capability. However, CGM can also be extended using derived classes in C++, allowing the geometric model to serve as the basis for other applications, for example mesh generation. CGM is supported on Sun Solaris, SGI, HP, IBM, DEC, Linux and Windows NT platforms. CGM also includes support for loading ACIS models on parallel computers, using MPI-based communication. Future plans for CGM are to port it to different solid modeling engines, including Pro/Engineer or SolidWorks. CGM is being released into the public domain under an LGPL license; the ACIS-based engine is available to ACIS licensees on request.

Intentionally Left Blank

Contents

1 Introduction

The Common Geometry Module (CGM) is a code library which provides geometry functionality used for mesh generation and other applications. This functionality includes that commonly found in solid modeling engines, like geometry creation, query and modification; CGM also includes capabilities not commonly found in solid modeling engines, like geometry decomposition tools and support for shared material interfaces. CGM is built upon the ACIS solid modeling engine, but also includes geometry capability developed beside and on top of ACIS. CGM can be used as-is to provide geometry functionality for codes needing this capability. However, CGM can also be extended using derived classes in C++, allowing the geometric model to serve as the basis for other applications, for example mesh generation. CGM is supported on Sun Solaris, SGI, HP, IBM, DEC, Linux and Windows NT platforms. CGM also includes support for loading ACIS models on parallel computers, using MPI-based communication. Future plans for CGM are to port it to different solid modeling engines, including Pro/Engineer or SolidWorks. CGM is being released into the public domain under an LGPL license; the ACIS-based engine is available to ACIS licensees on request.

This document is organized as follows. Section 2 describes the geometric model used in CGM, commonly referred to as a BREP model. Section 3 is a User's Guide, describing how CGM can be used as-is to support geometry functions without the addition of application-specific geometry capability. Section 4, the CGM Developer's Guide, describes the design of CGM in more detail, and how CGM capabilities can be extended using derived classes in C++. Section 5 gives details of CGM usage that are specific to the CUBIT Mesh Generation Toolkit; this section also serves as a useful example of how to extend CGM data structures for specific purposes. Appendix A describes licensing requirements for CGM, and for the ACIS-based parts of CGM. Appendix B contains source code listings for the C++ and C driver applications, including makefiles for building these applications.

This document is intended to serve as both an introduction to CGM as well as a reference. For those wanting a quick introduction to using CGM in its various modes, see Sections 3.6, 3.8 and 0; these sections describe CGM driver applications and do not require reading of this manual to understand the basic concepts.

2 Geometry Model

The basic geometry representation used in CGM is that of a Boundary Representation, or BREP. While this is a commonly used representation, the nomenclature used to describe it can vary greatly. Therefore, this section first introduces the basic elements of topology, and more importantly the terminology used to describe those elements in the rest of this document. In addition to topology, there are other important elements of the geometric model used in CGM that are important to understand; these elements are introduced in sections 2.2-2.4.

CGM is meant to serve as an interface to geometric models in a variety of formats, some of them solid model-based, some of them not. A discussion of the current and future support for various geometry representations is discussed in section 2.5.

2.1 Topology

The basic topological entities used in CGM, along with their dimensions, are shown in Table 1. Note that in the case of Edges and Faces, the most common nomenclature can be confusing in a finite element context, since there are also "edge" and "face" element types. In cases where the type of edge or face is unclear from the context, the terms "geometric" and "mesh" are used to qualify the term as pertaining to geometry or mesh, respectively.

In addition to the basic topological entities, there are a number of other geometric entities which can be used to describe the geometric model. These entities are typically not necessary for the complete geometric description of a BREP model, but they are useful constructs nonetheless. These entities are described in Table 2.

Table 1 "Basic" topological entities in the CGM geometric model.

Basic Topological Entity	Dimension
Vertex	0
Edge	1
Face	2
Volume	3

Table 2 Other geometric entities used in CGM geometric model.

	Entity	Description
Grouping Entities	Body	One or more volumes, share a transform and participate together in geometry transforms and booleans.
	Shell	Group of CoFaces describing a closed or open shell; if closed, describes a boundary of a volume. Can be inner or outer boundary.
	Loop	Group of CoEdges describing a boundary of a face. Can be inner or outer boundary.
Sense Entities	CoFace	Entity describing the orientation of a face as used in a shell, with respect to the face's normal.
	CoEdge	Entity describing the orientation of an edge as used in a loop, with respect to the edge's tangent direction.

In a typical application, a geometric model consists of one or more volumes, each volume bounded by one or more faces, with each face bounded by one or more edges and vertices. The entities in a BREP model are related to each other in a geometric hierarchy, which can be represented using a graph. In general, an entity of dimension d is bounded by one or more entities of dimension d-1, $0 < d \leq 3$.

There are several exceptions to the typical geometric model described above that are allowed in CGM models. First, CGM (and ACIS) allows curves to be bounded by a single vertex; these are called periodic curves, in reference to the periodic nature of the curve's parameter space. Likewise, periodic surfaces are also allowed; these are surfaces whose parameter space contains jump discontinuities, for example a cylindrical surface. Periodic entities can complicate applications like mesh generation; therefore, CGM can optionally split these entities into non-periodic entities.

In rare cases, geometry exists where an entity of dimension d is not strictly bounded by an entity of dimension d-1. The best example of this is a cone, where the apex is represented topologically by a vertex. Although there is no apparent curve corresponding to this vertex, CGM models this using a zero-length curve. Applications not desiring this behavior can request that the periodic entities in a body be split into non-periodic ones.

Geometry can be envisioned whereby a single basic topological entity bounds a higher order entity twice; an example would be a torus with a circular face contained twice in the bounding shell, or a "scratch" curve included twice on a surface. While these cases can be represented using judicious application of CoFaces or CoEdges, they are not typically found in real-world geometries. Cases involving surfaces included twice by a single shell are not strictly supported in CGM; however, edges can occur twice in a given loop, and these types of edges can occur with others on a loop (e.g. a re-entrant edge bounding a surface), or by themselves (e.g. a "scratch line" on a surface).

2.2 Non-manifold modeling

Non-manifold geometry is a general term describing models of mixed dimension, e.g. models containing volumes and dangling faces or edges. In the context of CGM, the term "non-manifold geometry" is used to indicate the presence of entities of dimension d-1 shared by more than one entity of dimension d. In

particular, “non-manifold” is often used in the context of CGM to describe groups of volumes which share faces in between them.

CGM represents shared edges and faces by actually representing a single edge or face, along with sense entities (CoEdge and CoFace, respectively) to describe the orientation of the entity with respect to the higher-dimension entity. For example, an edge can be part of two adjacent faces. The edge has a “natural” orientation, such that vertex V1 is the “start” of the edge and V2 is the “end”. Then, the two faces sharing that edge use the edge with opposite “sense”; face F1 uses the edge with “forward” sense, that is with a sense corresponding to the edge’s natural direction, while face F2 uses the edge with “reverse” sense. Likewise, in the case where two volumes share a common face, one volume uses the face with a forward sense while the other uses the face with a reverse sense. Note that multiple faces can share a given edge in a non-manifold model. However, in typical geometries, at most two volumes will share a given face, each with an opposite sense. If the senses are the same, overlapping geometry is usually indicated.

2.3 Groups

The grouping entities listed in Table 2 are specific to the kinds of geometry they contain; for example, bodies consist only of one or more volumes. In contrast, geometry groups are used in CGM to store one or more geometry entities of arbitrary type, including other groups. The entities stored in a given group typically have no implied relation to each other, and are grouped only for convenience, either for the user or for a specific application. For example, users may want to create a group of all entities falling on one side of a coordinate plane. Groups are used extensively in the CUBIT application of CGM, and in some cases inside CGM itself.

2.4 Attributes

Attributes are defined as information which can be associated with a particular geometric entity, but which are not intrinsic to the representation of that entity. For example, the name of a geometric entity, while not required to represent an entity, is associated directly to that entity. CGM provides functions for storing application-specific attributes directly on geometry entities, and functions for managing that data when geometry is stored to disk.

2.5 Geometric Representations Supported

CGM supports a variety of geometric representations; how it supports each also varies, between serving as an interface to some other code library and implementing that representation inside CGM.

Solid model geometry usually originates from a CAD system, for example SolidWorks or Pro/Engineer. There are also several commercially-available code libraries able to evaluate such geometry in its native format. CGM provides its best support for solid model-based geometry in the ACIS format [1]; this format can be exported directly from SolidWorks and other modelers, or can be translated into from various formats. Using components provided with the ACIS modeling engine, CGM can also import geometry in the IGES and STEP formats, though this mechanism of geometry transfer should be avoided if possible.

CGM provides facet-based geometric modeling, where a faceted shell can provide the description of the outer boundary(ies) of a solid. This mechanism is useful for representing scanned data like biological models, and for representing deformed models resulting from analysis. Faceted surfaces can be modeled as piecewise-planar surfaces, or can be used to compute a Bezier-type smoothed surface. The techniques used to compute this surface are described further in Ref. [2].

Another very powerful geometry modeling technique is that of “virtual geometry” [5]. In this modeling technique, the model topology can be modified while retaining the original geometric shape. For example, virtual geometry can be used to combine small topological surfaces together into a larger logical surface. This capability is used to support a variety of higher-level tools in the CUBIT mesh generation toolkit [1], including detail removal and mesh skew control. Virtual geometry works on any of the other supported formats in CGM.

Future plans include developing direct interfaces to other solid model-based formats. An interface has already been developed for the SolidWorks CAD system [3], though there are performance issues and this is a query-only interface. Several implementations of interfaces to Pro/Engineer's Granite interface have also been developed [4], but have more restricted distribution policies. Finally, there are other modelers for which the CGM team would like interfaces but do not have the time to develop. Since CGM is distributed under an open-source model (see Appendix A), there is a mechanism in place for sharing CGM source code for institutions desiring to collaborate on such an undertaking.

For more details about supported geometry formats in CGM, please contact Tim Tautges, tjtautg@sandia.gov.

3 User's Guide

3.1 Introduction

CGM encapsulates most of the geometry functionality required by the CUBIT Mesh Generation Toolkit; this includes interfaces to the ACIS solid modeling engine, but also incorporates non-manifold geometry, virtual geometry and other functions not found in ACIS or other solid modeling engines. CGM provides all the functions necessary to restore geometry to the state used to generate a CUBIT mesh; if geometry attributes are used to store the geometry after meshing in CUBIT, the geometric model can be restored in its final state with a few simple API calls. CGM can also be used to construct geometry from primitives or from fields of points, and evaluate this geometry for application-specific needs.

This User's Guide describes the use of CGM as-is to restore or create, and evaluate geometry. For information on extending the functionality of CGM, for example to store mesh directly on geometry, see the Developer's Guide in the following chapter. After describing the primary components in CGM and some implementation details, this chapter concludes with three example driver applications. Users looking for a quick-start guide to CGM should refer to these applications.

3.2 Module components

There are multiple components in CGM, distinguished by the functionality they implement and the interface used to access that functionality. The components fall under the general categories of tools, topology and geometry; these components are described in more detail in this section.

3.2.1 Geometry Tools and Engines

The primary interface to CGM is through its tools, and in particular through the GeometryQueryTool and GeometryModifyTool classes. GeometryQueryTool and GeometryModifyTool are implemented as singleton classes (see section 3.7), which implies that their functions can be viewed as API functions rather than functions associated with a geometry entity. GeometryQueryTool implements functions necessary for importing and exporting geometry files, and for querying the geometry, for example for surface normals and tangents, body volume, and closest point to an entity. GeometryQueryTool

is also the point of access for global lists of geometric entities. GeometryModifyTool provides functions for modifying geometry in various ways, including booleans, healing, and various types of decomposition. Separating geometry into query-only functions in GeometryQueryTool and non-query functions in GeometryModifyTool allows applications to link in only the needed functionality; this tends to reduce executable file sizes and improves modularity.

While the geometry tools provide a single interface to geometry capability, the actual representations are done using geometry engines. For example, AcisQueryEngine and AcisModifyEngine provide the functions needed to create, query and modify ACIS-based geometry. VirtualGeometryEngine and FacetQueryEngine/FacetModifyEngine provide virtual and facet-based representations, respectively.

Other tools in CGM, most of them also implemented as singleton classes, perform functions in specific areas, for example MergeTool, which changes manifold to non-manifold geometry, and FeatureTool, for performing automatic decomposition.

3.2.2 Topology

There are various topology traversal functions that are accessible from any of the CGM topological entities; these functions give all the topological entities of a given type connected to an entity, e.g. all the vertices in a volume, or all the bodies containing an edge. Most topology traversal functions are called as member functions of an entity object.

3.2.3 Geometry

Functions which are used to evaluate the geometric representation of an entity are called as member functions of that entity. For example, there are member functions for an edge which move a given coordinate position to the closest point on that edge, or which return the length of the edge.

3.3 Tools and Engines Component

The primary access point to geometric capability in CGM is through GeometryQueryTool and GeometryModifyTool, each of which has one or more underlying engines. Other tools provide functionality which creates or modifies geometry. For example, MergeTool detects coincident entities and combines them to form non-manifold geometry. The most commonly used tools are described in more detail in this section.

3.3.1 Geometry Engine Initialization

CGM requires the calling application to create geometry engines of the types required for the application. This allows the selection of geometry engine to happen at application compile time, and allows applications to all use the same set of CGM libraries regardless of the engines used in each application. Geometry engines can be created before or after CGM itself is initialized (this is described in Section 3.7.1). Three of the most common combinations are:

- Query-only, ACIS-based In this case, an AcisQueryEngine object is created by calling its instance function in the application:

```
AcisQueryEngine *aqe = AcisQueryEngine::instance();
```

  Inside the constructor of AcisQueryEngine, the created engine is passed to the GeometryQueryTool singleton instance (which gets instantiated during this process, if one has not yet been created).

- Modify, ACIS-based
  In this case, the application requires functionality to create and/or modify geometry. This is accomplished by creating an AcisModifyEngine instance:

```
AcisModifyEngine *ame = AcisModifyEngine::instance();
```

  During this process, an AcisQueryEngine will be created, if it does not yet exist, and both will be passed to their respective geometry tools (GeometryModifyTool and GeometryQueryTool, respectively). Note that there is dependence from AcisModifyEngine on AcisQueryEngine, but not vica versa; this allows the creation of an AcisQueryEngine without creating an AcisModifyEngine, which saves space in the application executable.

- Query-only, facet-based
  In certain cases, applications need to construct and query facet-based geometry, for example for remeshing deforming models. In this case, a FacetQueryEngine must be created:

```
FacetQueryEngine *fge = FacetQueryEngine::instance()();
```

Note that this does not require the use of ACIS, which results in substantial savings in executable size.

In all cases, no re-compilation of GeometryQueryTool or GeometryModifyTool is required to support these various functionalities. This simplifies the maintenance of CGM and supports its use in many different contexts.

After the initialization of one type of engine (query-only or non-query, ACIS-based or facet-based), other engines can be created, and will be added to the corresponding tools as required. That way, an application can start functioning in a query-only mode and move to a non-query mode if/when necessary.

3.3.2 Geometry Query Operations: GeometryQueryTool

Once the appropriate query engine(s) are instantiated, all geometry queries should be done through GeometryQueryTool or the specific topology classes. This ensures that the query will be made to the appropriate engine (ACIS, facet, etc.), and also allows the implementation of engine-independent code once rather than in every engine. Some of the common types of query functionality are described below. In all cases, functions in GeometryQueryTool are accessed through a singleton instance:

```
GeometryQueryTool::instance()->import_solid_model(…)
```

3.3.2.1 Geometry import/export

The most common method for defining geometry with CGM is to import it from an ACIS .sat file. These files can be used to save geometry directly from CGM, and can also be written directly from several CAD tools including SolidWorks and Pro/Engineer. The functions used to import and export geometry are listed in Table 3. Only the required arguments are listed; each of the functions defines several optional arguments, which can be used for example to define a log file to which any informational messages are written.

Table 3 GeometryModifyTool functions for importing and exporting geometry from/to a disk file.

Function	Purpose
`GeometryQueryTool::import_solid_model`	Import a solid model file of a specified type (ACIS_SAT, ACIS_SAB, etc.)
`GeometryQueryTool::export_solid_model`	Export entities to a file of a specified type.

3.3.2.2 Global entity lists

After geometry is imported into CGM, it is maintained in global lists accessible through GeometryQueryTool. Individual basic topological entities are identified primarily using integer ids, but can also be identified using names (which are either defined by the application or can be assigned to default values). Id numbers are unique within a given entity type, while names are unique across all entities (this behavior can be modified, but this is discouraged).

In addition to retrieving individual entities, there are functions in GeometryQueryTool which give access to the global entity lists. There are functions which directly access and step through lists, and functions which pass back copies of the global lists for use in the application code. The functions used to access geometry are summarized in Table 4.

Table 4: GeometryQueryTool functions for accessing global entity lists.

Function	Description
`GeometryQueryTool::cubit_entity_list`	Append entities of the specified type to a DLCubitEntityList
`GeometryQueryTool::ref_entity_list`	Append entities of the specified type to a DLRefEntityList

`GeometryQueryTool::bodies`
`GeometryQueryTool::ref_xxx`
`(xxx = volumes, faces, edges, vertices, groups, parts, assemblies,`
	Append all bodies in the model to a DLBodyList
	Similar for ref_xxx

coordsys)	
GeometryQueryTool::get_ref_entity	Return an entity of a given type and id
GeometryQueryTool::get_body **GeometryQueryTool::get_ref_xxx** **(xxx = volumes, faces, edges, vertices, groups, parts, assemblies, coordsys)**	Return a body of a given id Similar for get_ref_xxx
GeometryQueryTool::num_bodies **GeometryQueryTool::num_xxx** **(xxx = volumes, faces, edges, vertices, groups, parts, assemblies, coordsys)**	Return the number of bodies in the global list Similar for num_ref_xxx
GeometryQueryTool::get_first_body **GeometryQueryTool::get_first_ref_xxx** **(xxx = volumes, faces, edges, vertices, groups, parts, assemblies, coordsys)** **GeometryQueryTool::get_next_body** **GeometryQueryTool::get_next_ref_xxx** **GeometryQueryTool::get_last_body** **GeometryQueryTool::get_last_ref_xxx**	Return the first body in the global list, positioning the list at the beginning; similar for get_first_ref_xxx. For get_next_xxx, step the list and return that item. For get_last_xxx, get the last item in the list and position the list at the end.

3.3.2.3 Geometric Queries

The primary purpose of having geometric capability in an application is to provide geometric information about a model, including geometric extent (length, area, etc.), geometric location (closest point), and other data (surface normals, tangents). Most of the geometric query functions in CGM are accessed through the individual entities like curves, surfaces and volumes. Selected functions are listed in Table 5.

Table 5: Geometric query functions accessed through individual entities.

RefEntity::measure()	Generic geometric extent function, returns the length/area/volume of the edge/face/volume and 1.0 for vertices.
RefEdge::position_from_u(…)	Forward-evaluate a curve based on a u coordinate.
RefFace::normal_at(…)	Return the face normal at a given point.
RefFace::move_to_surface(…)	Move a point to the closest point on the (untrimmed) surface
RefVolume::point_containment(…)	Returns whether a given point is inside, outside, or on the boundary of a volume

3.3.3 Geometry Modify Operations: GeometryModifyTool

Geometry modification functions are separated into another tool, GeometryModifyTool. This tool interacts closely with GeometryQueryTool, but is separate to allow use of GeometryQueryTool alone. Functions in GeometryModifyTool are accessed through a singleton instance:

```
CubitStatus status = GeometryModifyTool::instance()->intersect(…)
```

3.3.3.1 Geometry Primitives

Geometry is often created using a well-known primitive, e.g. bricks, cylinders and spheres. The geometry primitive functions provided in GeometryModifyTool are listed in Table 6 (see the class header files for complete descriptions).

Table 6 Geometry primitives and functions in GeometryModifyTool.

Function	Description
GeometryModifyTool::brick	Brick with dimensions specified in x, y and z.

`GeometryModifyTool::cylinder`	Cylinder of specified height and radius; can create elliptic cylinder or conic using minor radius and top radius, respectively.
`GeometryModifyTool::sphere`	Sphere with specified radius.
`GeometryModifyTool::prism`	N-sided prism with specified height (N >= 3).
`GeometryModifyTool::torus`	Torus with major and minor radii specified.
`GeometryModifyTool::pyramid`	Same as prism.

3.3.3.2 Geometry Creation

In addition to primitives, geometry can be created by creating vertices, then edges, then faces, etc. These functions are summarized in Table 7.

Table 7 GeometryModifyTool functions for geometry creation.

Function	Description
`GeometryModifyTool::make_RefVertex`	Make a vertex from a point.
`GeometryModifyTool::make_RefEdge`	Make an edge from two vertices, optionally on a face, along a list of segments, as an ellipse/parabola/logarithmic spiral
`GeometryModifyTool::make_RefFace`	Make a face from a list of bounding edges, as a plane/sphere/cone/spline/best fit surface.
`GeometryModifyTool::create_body_from_surfs`	Make a volume from a list of bounding faces.
`GeometryModifyTool::make_Body`	Make a body from a list of volumes.

In addition, bodies can be created by sweeping 2-dimensional faces into the third dimension. Since this is a common solid modeling operation, CGM provides several functions of this type; these functions are summarized in Table 8. Complete syntax and descriptions of these functions are located in the class header files.

Table 8 GeometryModifyTool functions which sweep a face into a 3-dimensional solid.

Function	Description
`GeometryModifyTool::sweep_translational`	Sweep a face into a volume by translation.
`GeometryModifyTool::sweep_rotational`	Sweep a face into a volume by rotation.
`GeometryModifyTool::sweep_along_curve`	Sweep a face into a volume by following a specified curve.

3.3.3.3 Transformations

Transformation functions typically take a body and transform its geometry, for example by scaling, rotation, or reflection. Since they involve local changes to the body geometry, transformation functions are called primarily using the Body pointers. The exception to this rule is the reflect function, which is called using GeometryModifyTool and which yields a new Body. Transformation functions available on Body and GeometryModifyTool are listed in Table 9. Since some of these functions are accessible through the Body class, they can be used even in a query-only setting; in this case, no extra cost is incurred in the size of executables due to this capability.

Table 9 Transformation functions in the Body and GeometryModifyTool classes.

Function	Description
`GeometryQueryTool::reflect`	Reflect a body across a given plane.
`GeometryQueryTool::translate`	Move a body by a given dx, dy and dz.
`GeometryQueryTool::scale`	Scale a body by a given factor.
`GeometryQueryTool::rotate`	Rotate a body about a given axis, vector or curve.
`GeometryQueryTool::restore_transform`	Restore a body to the state before the previous

	transformation.

3.3.3.4 Booleans

Booleans are set operations on two or more bodies, which yield two, one or no bodies as a result, depending on the type of and success of the operation. Boolean functions, as implemented in GeometryModifyTool, also return a status value, which should be checked after the functions are called. Boolean functions provided by GeometryModifyTool are summarized in Table 10. This list is not exhaustive, but contains the most commonly used boolean functions; a complete list can be found in the class header files.

Unite, subtract and intersect are similar to the corresponding set operations, and require no further explanation. Imprint operations, as the name implies, imprints topology from one body onto adjacent bodies; by definition, the imprinted topology is not geometrically significant, i.e. it does not change the volume of a body. Imprinting is done to ensure that adjacent surfaces share identical topology, which enables them to be merged together. Regularize is the opposite of imprint; this function removes topology from a body which is not geometrically significant.

Table 10 Boolean functions in GeometryModifyTool.

Function	Description
`GeometryModifyTool::unite`	Unite the given body or bodies into a single body.
`GeometryModifyTool::subtract`	Subtract one or more bodies from a given body.
`GeometryModifyTool::intersect`	Intersect one or more bodies from a given body, or intersect pairwise.
`GeometryModifyTool::imprint`	Imprint two or more bodies together.
`GeometryModifyTool::regularize_body`	Regularize one or more bodies (opposite of imprint).

3.3.3.5 Geometry Decomposition Functions

Geometry decomposition is an important part of a hexahedral meshing application, and must be supported well by a geometry module. CGM provides many functions for decomposing geometry. Typically, a cutting surface is specified using an existing surface, a surface extending from an existing surface, or a semi-infinite surface like a coordinate plane or cylindrical surface; or, sometimes an entire body is used as a cutting tool. The cutting tool or surface along with the body or bodies to be cut are passed to a specific webcut function, which returns the results of the cutting operation. GeometryModifyTool functions supporting geometry decomposition are summarized in Table 11.

Table 11: Functions supporting geometry decomposition.

Function	Description
`GeometryModifyTool::webcut_with_xxx (xxx = plane, sheet, vertices, cylinder, surface, extended_surface, body)`	Decompose one or more bodies with a given geometry entity or implicitly-defined geometric entity. Keeps all bodies resulting from decomposition.
`GeometryModifyTool::section`	Decompose one or more bodies with a planar surface or coordinate axis. Keeps bodies on one side of the surface only.
`GeometryModifyTool::split_body`	Changes multi-volume body into several single-volume bodies.
`GeometryModifyTool::split_periodic`	Splits periodic surfaces on one or more bodies into multiple surfaces.

3.3.4 MergeTool

Non-manifold geometry, in the context of CGM, simply means geometry containing vertices, edges and faces which can be shared by more than one volume or by multiple free faces or edges. Solid models are created and imported as manifold models; a merge operation must be performed on a model to convert it to a non-manifold representation. The merge operation simply searches for geometry entities of like topology and geometry (within a specified tolerance), and merges any it finds. Merging is an important part of applications built on CGM, and care should be taken to merge models requiring contiguous regions before interacting with those models.

Typically, geometry coming straight from a CAD system will require some cleaning up before it will merge correctly (i.e. before the only surfaces remaining in a multi-material model are surfaces on the "outside" of the part). Geometry can be cleaned up by importing it into CUBIT or another application and performing the necessary geometry manipulations. At a minimum, all bodies should be imprinted on one another to guarantee that entities which are coincident in space also have like topology. While this is most often done in another application before saving the final geometry, it might also be necessary to imprint the bodies in the CGM application.

Functions which control merging are summarized in Table 12.

Table 12: MergeTool functions which control merging of manifold into non-manifold geometry.

Function	Description

`MergeTool::merge_all_bodies`
`MergeTool::merge_all_refxxx`
`(xxx = vertices, edges, faces)`
	Performs merge check on all entities of the specified type.

`MergeTool::merge_bodies`
`MergeTool::merge_entities`
`MergeTool::merge_refxxx`
`(xxx = vertices, edges, faces, volumes)`
	Perform merge check on entities passed in as arguments.

`MergeTool::contains_merged_entities`	Returns CUBIT_TRUE if one or more bodies passed in contains merged child entities.
`MergeTool::entity_merged`	Returns CUBIT_TRUE if the entity passed in is merged.
`MergeTool::merge_has_occurred`	Returns CUBIT_TRUE if a merge has occurred anywhere in the model.

3.3.5 VirtualGeometryEngine

When a solid model is imported or created based on the ACIS geometry engine, each entity in the ACIS model has a corresponding entity in the CUBIT model. For example, each FACE in the ACIS model will have a corresponding RefFace in CUBIT. Sometimes, it is desirable to change the CUBIT model without modifying the underlying ACIS model; VirtualGeometryEngine provides this capability.

VirtualGeometryEngine provides functions for combining or splitting edges, faces and volumes. After these operations, the CUBIT model reflects the new geometry, while the ACIS model remains unchanged. This capability can be used, for example, to combine many very small surfaces into a larger topological surface.

Table 13 summarizes the functions for creating, removing, and performing other interactions with virtual geometry.

Table 13: VirtualGeometryEngine functions for creating, removing and otherwise interacting with virtual geometry.

Function	Description
`PartitionTool::partition`	Partition the given entity using explicit or implicit geometry (implicit geometry is geometry whose geometric data are specfied by the user; explicit geometry is a geometry entity

	already extant in the model)
PartitionTool::unpartition	Remove the partition on one or more entities; removes only one partition.
PartitionTool::unpartitionAll	Remove all partitions on one or more entities, passing back the list of restored entities.
CompositeTool::composite	Composite the given list of entities into a single entity.
CompositeTool::uncomposite	Remove composite, passing back list of restored entities.
CompositeTool::isComposite	Returns CUBIT_TRUE if the entity passed in is a composite entity.
CompositeTool::okayToComposite	Returns CUBIT_TRUE if the entities passed in can be composited together.

3.4 Geometry Component

3.4.1 Geometric representation: TopologyBridge

Before any merging operations, the geometric topology in CGM corresponds exactly to that of the solid model. In the CGM datastructure, each solid model entity has a corresponding TopologyBridge object, which is unique to that solid model entity. Each RefEntity points to a single TopologyBridge (through a BridgeManager object, the purpose of which is explained later). Thus, before merging, there is a one-to-one correspondence between each RefEntity and its corresponding solid model entity.

The TopologyBridge object for a RefEntity is really the geometric representation of that entity. All geometric queries to a RefEntity (e.g. the length of an edge, or the distance between a point and the closest point on a face) are passed directly to the TopologyBridge object, which forwards the request to the solid model entity. In this way, the solid modeling engine is used for geometric queries, rather than duplicating any data and code necessary to implement those queries in CGM. The public interface to geometric queries is implemented in the RefEntity objects instead of through TopologyBridge in order to reduce the number of objects applications have to keep track of.

3.4.2 Alternative geometric representations

CGM has been designed to allow alternative representations of geometry; indeed, this capability is being developed actively in the virtual and facet-based geometry components. Alternative geometric representations can be used simply by providing code underneath TopologyBridge that provides the evaluation functions required by the corresponding RefEntity. See the VirtualCurve class for an example of such an alternate representation. This design also simplifies the process of substituting another solid modeling engine for ACIS; a PROE-based implementation of CGM is being developed which utilizes these design features, and other implementations are being considered.

3.5 Topology Component

Geometric models consist of topological entities like vertices, edges and faces, related to one another through a topology graph. Each entity of dimension d is bounded by one or more entities of dimension d-1, and, most of the time, bounds one or more entities of dimension d+1. One of the most common operations while querying geometry is to traverse this topology graph, finding entities of dimension m related to one or more entities of dimension n. CGM provides many functions for topology traversal.

Most topology traversal functions are called as member functions from the entity from which the traversal starts; the functions are implemented in TopologyEntity. One exception to this is the get_related_entity functions, which take as input a list of entities and return a list of related entities of a specified type. In all cases, if a function is called to return all entities of the same type as the entity from which the function was called, that entity is returned in the list.

The topology traversal functions are summarized in Table 14.

Table 14: Topology traversal functions implemented in TopologyEntity.

Function	Description

`TopologyEntity::bodies`
`TopologyEntity::ref_xxx (xxx=volumes, faces, edges, vertices)`
`TopologyEntity::shells`
`TopologyEntity::loops`
`TopologyEntity::co_faces`
`TopologyEntity::co_edges`
	Return list of bodies related to the entity from which the function was called.
	Similar for ref_xxx functions.
	Similar for shells, loops, co_faces, co_edges functions.

For example, the following call is used to find all the vertices connected to volume pointer vol1 and store them in list vertex_list:

```
CubitStatus return_status = vol1->ref_vertices(vertex_list);
```

Sometimes, it is desirable to gather the topological entities of the next higher or lower dimension, without determining what the specific type of those entities is. For example, for most of the meshing algorithms in CUBIT, before meshing an entity of dimension d, all the bounding entities of dimension d-1 must already be meshed. Topology traversal functions which return parent and child entities without requiring the parent or child entity type as input are used for this purpose and are defined in RefEntity.

The notion of a parent or child entity of dimension d+1 or d-1 only makes sense for our basic topology types vertex, edge, face, volume and body. Since we do not require a target entity type as input, these traversal functions either give immediate parent or child entities, or they give a list of all parent or child entities.

The topology traversal functions defined in RefEntity are summarized in Table 15. Like the traversal functions in TopologyEntity, these functions are called as member functions from a RefEntity object.

Table 15: Parent and child topology traversal functions defined in RefEntity.

Function	Description
`RefEntity::get_child_ref_entities`	Return a list of all immediate children of the RefEntity.
`RefEntity::get_parent_ref_entities`	Return a list of all immediate parents of the RefEntity.
`RefEntity::get_all_child_ref_entities` `RefEntity::get_all_parent_ref_entities`	Return a list of all children, traversing down to dimension d=0 (i.e. vertices); similar for parents, but traversing up to dimension d=4 (i.e. bodies).
`RefEntity::get_child_ref_entity_type` `RefEntity::get_parent_ref_entity_type`	Return the EntityType of the the immediate child RefEntity's. Similar for get_parent_entity_type.
`RefEntity::is_child` `RefEntity::is_parent`	Return CUBIT_TRUE if the input entity is a child/parent (immediate or not) of the RefEntity

There are additional topology traversal functions defined in BasicTopologyEntity, SenseEntity and GroupingEntity; these functions generally provide traversals to the next higher or lower type of TopologyEntity, and are not as commonly used as the traversal functions in TopologyEntity and RefEntity. Descriptions of these functions are located in the automatic documentation for those classes.

3.5.1 Non-manifold modeling: BridgeManager

CGM accomplishes non-manifold modeling not by merging actual solid model entities, but by having a single RefEntity correspond to multiple solid model entities. Merging takes entities with like topology and geometry (within a geometric tolerance), and combines them into a single entity. The changes to the datastructure resulting from this operation can be described with a simple example, merging two edges together. First, the two RefEdges are compared, and the one with the lowest id is designated the "keeper"; the other RefEdge is designated the "dead" entity. The TopologyBridge object for the dead entity is added

to the list of TopologyBridge objects on the keeper's BridgeManager. Then, the topology graph is changed such that all parents of the dead entity point to the keeper entity instead. The sense entity immediately above is changed to incorporate the correct sense (this process is explained in more detail below). Finally, the dead entity, along with its TopologyBridge object, are deleted. We are left with the keeper RefEdge and its BridgeManager object, which maintains a list of two TopologyBridge objects. The process for merging more than two entities together is similar, but now there are more than two TopologyBridge objects in list in BridgeManager.

There are a few subtle things to note about the datastructure for non-manifold topology. First, we assume that lower order topology has already been merged (this is checked in MergeTool and accomplished during the merging process); after fixing the topology graph for the parents of the dead entity, the topology graph can be traversed as before. After merging takes place, from the CGM model point of view, there is a single shared entity. Any data assigned the shared entity, either by CGM (e.g. names, ids) or in an application (e.g. mesh), applies to both the underlying solid model objects.

The sense entities immediately above the objects being merged may or may not be changed, depending on the type of entity and it's actual geometry. Consider first merging two edges together (see). Edge 1 is part of loop 1, which uses the edge with a "forward" sense, and similarly for edge 2 in loop 2. When edges 1 and 2 are merged together, edge 2 goes away and loop 2 now uses edge 1. Note though that loop 2 now uses edge 1 with a "reverse" sense; this is stored in the datastructure by modifying coedge 2, which maintains the connection between the loop and the edge. Consider next merging three edges together (see). Loops 2 and 3 now use edge 1; however, only coedge 2 must have its sense changed; the sense of coedge 3 is unchanged.

Merging faces is very similar. However, it is rare that two faces are merged together and the corresponding cofaces both maintain the same sense; this would correspond to two volumes occupying the same physical space, which is usally not done. Therefore, MergeTool prints a warning when merging two faces like this.

The code which checks and changes coedge and coface senses if necessary is located in MergeTool.

3.6 Building Applications

The previous sections in this User's Guide describe in more detail the design and use of the geometry datastructure in CGM. In this section, the steps necessary to build applications which use CGM are described.

3.6.1 Overall directory structure

CGM and its components are arranged in a directory structure designed to minimize unnecessary dependencies between components. This directory structure is depicted in Table 16. In order to compile applications using CGM, both directories, as well as any subdirectories, must be available at compile time. The purpose of the code in each subdirectory is also described in Table 16.

Table 16: CGM directory structure and purpose of code in each subdirectory.

Directory	Purpose	Depends on:
$(CUBIT_BASE_DIR)/util	Utility functions for other code in CGM	(none)
$(CUBIT_BASE_DIR)/util/OtherFiles	Configuration files for compiling CGM on various platforms	(none)
$(CUBIT_BASE_DIR)/geom	Core CGM datastructure and tool classes.	util, list, virtual, facet
$(CUBIT_BASE_DIR)/geom/ACIS	CGM interface to ACIS (available to ACIS licensees; see Appendix A).	geom, util, list, ACIS package
$(CUBIT_BASE_DIR)/geom/virtual	Virtual geometry component (datastructure and tools).	geom, util, list
$(CUBIT_BASE_DIR)/geom/facet	Facet geometry component (datastructure and tools).	geom, util, list

3.6.2 Compiling applications

A sample application is given in the following section; the makefile associated with this driver application should be used as a reference for developing new applications based on CGM. Parts of this makefile can be inserted into the makefile of existing applications that want to begin using CGM. This section gives some additional notes on compiling applications based on CGM.

As noted in Table 16, some of the code in CGM depends on the ACIS solid modeling engine; however, it is not necessary to link CGM applications with ACIS if ACIS-based geometry will not be used.

3.7 Implementation Notes

Included in this section are notes about the general implementation of CGM that should be known to applications developers.

3.7.1 Initializing CGM

CGM is initialized by calling the function:

```
CGMApp::instance()->startup(argc, argv);
```

Before or after initializing CGM, geometry engines required by the application should be created using the procedure described in Section 3.3.1. Separating the creation of geometry engines from basic CGM initialization allows the use of the same common CGM libraries with different combinations of engines.

3.7.2 Singleton tools

Most tools accessed through the public interface of CGM are implemented as singleton classes. This well-known design pattern is used for implementing functionality that is of a global nature, without requiring the functions themselves to be global (see [1] for a detailed description of the singleton and other design patterns). Singleton classes typically have constructors that are private, which prevent the creation of new singleton objects by classes other than the singleton class itself. Singleton class objects are accessed using an instance function:

```
MyTool *tool = MyTool::instance();
```

This function is static, so it does not require an object of type MyTool before it is called. Inside the instance function, a singleton object of type MyTool is created if it does not already exist, and a static pointer to this tool is maintained in the class. This pointer is then returned to the calling application.

Singleton classes are also used to accomplish extensibility of CGM. CGM classes and tools call singleton instances to accomplish things like create geometry and geometry attributes. Applications can substitute their own implementations of these singleton tools, which in effect allows the application-specific code to be called from CGM without CGM being dependent on the applications. This requires some careful initialization of CGM from applications which extend its capabilities. This process is described in more detail in the following chapter.

3.7.3 Creation/modification through GeometryModifyTool

As stated earlier, all geometry creation and modification is accomplished by calling functions in GeometryModifyTool. Although very few of these functions are actually implemented in GeometryModifyTool, this class provides a focal point for calling geometry functions, reducing the size of the CGM interface applications developers need to understand before using the package. In general, developers should look for needed functionality first in GeometryModifyTool.

3.7.4 Geometric and Topological Queries

Certain functions are clearly object-oriented functions which are associated directly with geometry objects. Examples include functions which return the length of an edge, the vertices connected to a volume, or the closest point to a given surface. Object-oriented functionality like this is usually implemented in the

RefEntity leaf classes (RefVertex, RefEdge, etc.), or in their parent classes (BasicTopologyEntity or RefEntity). These implementations either pass the request on to another class for actual implementation, or the functions are defined and implemented in one of the parent classes. In general, functions implemented on RefEntity objects are query-only; functions that modify geometry are typically implemented in tools like GeometryModifyTool or MergeTool.

3.7.5 Geometry Attributes

Functions which implement geometry attribute capabilities can be called directly from RefEntity objects; these functions are actually implemented in CubitAttribUser, a parent class of RefEntity. By default, only entity name attributes are saved and restored automatically. To enable automatic storing and retrieval of geometry attributes, set a parameter using the function call:

```
CGMApp::instance()->attrib_manager()->auto_flag(CUBIT_TRUE);
```

This function sets options for all the attribute types such that they are written and read automatically.

The geometry attributes capability can be used to store application-specific information on geometric entities. The mechanism used to implement application-specific attributes is described in the next chapter.

3.8 C++ Driver Application

In this section, a simple CGM driver application is described. This application, called mergechk, imports one or more geometry files, computes any overlaps between the bodies, imprints the bodies, then merges geometry into non-manifold geometry. Although very straightforward, this application demonstrates how to import geometry, perform boolean operations on it, then further query the geometry. The source code for this application is shown in Table 17 - Table 21. Some declarations and comments in the driver code have been removed for brevity; the complete mergechk application is distributed with the CGM libraries, in the cgm_apps/examples/driverc++ subdirectory.

Each of the code sections in mergechk is described below.

3.8.1 Forward declarations and main

The source code for forward declarations and the main function are shown in Table 17. This file begins with the inclusion of files containing declarations of functions and datastructures. In most CGM applications, GeometryQueryTool.hpp and/or GeometryModifyTool.hpp will be included, since these are the primary points of access to CGM. In the main function, the first few lines of code initialize first CGM (by calling CGMApp::instance()->startup()), then the AcisModifyEngine, which itself initializes AcisQueryEngine and GeometryModifyTool. Even in cases where the AcisModifyEngine object is not used immediately, it should be created before any other CGM functions are called, since the constructor initializes many static datastructures used by CGM.

After CGM initialization, the application calls several functions which import the geometry file(s), evaluate any intersections, then perform imprint and merge operations on the bodies. Results are then printed out for the user.

In this application, since it exits immediately after these geometry calculations, there is no need to explicitly shut down CGM.

Table 17: Pseudo code for C++ driver code, forward declarations and main.

```
// include tool and datastructure declarations
#include “GeometryModifyTool.hpp”
#include “GeometryQueryTool.hpp”
(…)

/// main program - initialize, then send to proper function
int main (int argc, char **argv)
```

```
{

    // initialize CGM
  CGMApp::instance()->startup();

    // create an ACIS modify engine (query engine created by that)
  AcisModifyEngine::instance();

    // Read in the geometry from files specified on the command line
  CubitStatus status = read_geometry(argc, argv);

    // Check for overlaps
  status = evaluate_overlaps();

    // Imprint bodies together, reporting on results
  status = imprint_bodies();

    // Merge bodies
  status = MergeTool::instance()->merge_all_bodies();

    // Print number and ids of non-shared surfaces
  status = print_unmerged_surfaces();
}
```

3.8.2 Reading geometry files

In the read_geometry() function, shown in Table 18, each file is opened, then the file pointer is passed to GeometryQueryTool::import_solid_model function. This function reads all the geometry entities defined in that file and stores them in the CGM database. These entities are accessed through other functions in GeometryQueryTool.

The return value from import_solid_model() should be checked for indications of problems reading the geometry file.

Table 18: Pseudo code for C++ driver; read_geometry() function.

```
CubitStatus read_geometry(int num_files, char **argv)
{
  GeometryQueryTool *gqt = GeometryQueryTool::instance();

    // For each file, open and read the geometry
  for (i = 1; i < num_files; i++) {
    file_ptr = fopen(argv[i], "r");
    if (file_ptr == NULL) PRINT_ERROR("Could not open file %s\n", argv[i]);
    else {
      status = gqt->import_solid_model(file_ptr, argv[i], "ACIS_SAT");
    }
  }
  return CUBIT_SUCCESS;
}
```

3.8.3 Evaluating overlaps using pairwise intersections

To find overlaps, each body is intersected with all the other bodies in the model; this is implemented in the evaluate_overlaps() function, shown in Table 19. A copy of the body list is retrieved from CGM using the GeometryQueryTool::bodies() function. Intersections are checked by removing a body from the list and intersecting it with all the bodies remaining on the list. An option is passed to the intersect function instructing CGM to keep the old bodies being intersected, since these bodies must be checked for merging later in the application.

If there are no overlaps, there should be no geometry entities resulting from the overlap calculation. If there are entities remaining, these are reported to the user. These entities are reported by topology type, with the topology-type filtering done using a list cast operation.

GeometryQueryTool::delete_Body() is called at the end of evaluate_overlaps() to delete any bodies produced during the intersection operation.

Table 19: Pseudo code for C++ driver; evaluate_overlaps() function.

```
CubitStatus evaluate_overlaps()
{
    // evaluate overlaps by intersecting bodies pairwise
  GeometryModifyTool *gmt = GeometryModifyTool::instance();

    // make a copy of the body list for use in this function
  DLIList<Body*> all_bodies, all_new_bodies;
  GeometryQueryTool::instance()->bodies(all_bodies);

    // step forward on this list, extracting the first body and using it
    // as a tool for remaining bodies
  for (i = all_bodies.size(); i > 0; i--) {
    all_bodies.last();
    Body *tool = all_bodies.remove();

      // intersect the tool with remaining bodies; make sure and keep old
      // bodies, since we're not using copies; save new bodies for evaluation
      // later
    DLIList<Body*> new_bodies, temp_bodies = all_bodies;
    CubitStatus status = gmt->intersect(tool, temp_bodies, new_bodies,
                                        CUBIT_TRUE);
    all_new_bodies += new_bodies;
  }

    // count number of geometric entities in new bodies; if there are no
    // overlaps, this number should be zero
  DLIList<RefEntity*> child_entities, temp_children;

    // first get all child entities of the new bodies
  for (i = all_new_bodies.size(); i > 0; i--) {
    all_new_bodies.get_and_step()->get_all_child_ref_entities(temp_children);
    child_entities += temp_children;
    temp_children.clean_out();
  }

    // then filter the list, keeping only unique entities
  temp_children.clean_out();
  temp_children.merge_unique(child_entities);

    // now report
  if (temp_children.size() != 0) {

      // check vertices
      // temp_entities is cleaned out in the CAST_LIST macro, so no need
      // to do that here
    DLIList<RefEntity*> temp_entities;
    CAST_LIST(temp_children, temp_entities, RefVertex);
    if (temp_entities.size() > 0)
      PRINT_INFO("   Vertices: %d\n", temp_entities.size());
…
      // now delete all the bodies produced by the intersections
    DLIList<Body*> new_bodies;
    CAST_LIST(temp_children, new_bodies, Body);
    for (i = new_bodies.size(); i > 0; i--)
```

```
      GeometryQueryTool::instance()->delete_Body(new_bodies.get_and_step());
  }

    // we're done
  return CUBIT_SUCCESS;
}
```

3.8.4 Imprinting bodies

Bodies are imprinted in mergechk in the imprint_bodies() function, shown in Table 20. Geometry entities can merge together only if they have like topology and geometry. Typically, models coming from CAD packages like Pro/Engineer or SolidWorks do not have topology imprints on neighboring bodies. Imprinting is accomplished using the GeometryModifyTool::imprint() function. Since imprint will always change the number of topological entities in a model, numbers of entities before and after the imprint can be compared to determine whether any imprints were made.

Table 20: Pseudo code for C++ driver; imprint_bodies() function.

```
CubitStatus imprint_bodies()
{
    // imprint all the bodies together, and report number of new
    // entities formed
  GeometryQueryTool *gqt = GeometryQueryTool::instance();

    // first, count old entities
  int num_vertices = gqt->num_ref_vertices();
  int num_edges = gqt->num_ref_edges();
  int num_faces = gqt->num_ref_faces();
  int num_volumes = gqt->num_ref_volumes();
  int num_bodies = gqt->num_bodies();

    // imprint the bodies together, discarding old bodies
  DLIList<Body*> old_bodies, new_bodies;
  gqt->bodies(old_bodies);
  GeometryModifyTool::instance()->imprint(old_bodies, new_bodies);

    // now count new numbers of entities, subtracting old numbers
  num_vertices = gqt->num_ref_vertices() - num_vertices;
  num_edges = gqt->num_ref_edges() - num_edges;
  num_faces = gqt->num_ref_faces() - num_faces;
  num_volumes = gqt->num_ref_volumes() - num_volumes;
  num_bodies = gqt->num_bodies() - num_bodies;

    // report results
  if (!num_vertices && !num_edges && !num_faces && !num_volumes && !num_bodies)
    PRINT_INFO("Imprinting resulted in no new entities.\n");
  else {
    PRINT_INFO("Imprinting resulted in the following numbers of new
entities:\n");
    if (num_vertices) PRINT_INFO("   %d vertices.\n");
    if (num_edges)    PRINT_INFO("   %d edges.\n");
    if (num_faces)    PRINT_INFO("   %d faces.\n");
    if (num_volumes)  PRINT_INFO("   %d volumes.\n");
    if (num_bodies)   PRINT_INFO("   %d bodies.\n");
  }

    // ok, we're done
  return CUBIT_SUCCESS;
}
```

3.8.5 Merging and printing results

In the main function, after the call to imprint_bodies() (see Table 17), all bodies in the model are checked for potential merges by calling MergeTool::merge_all_bodies(). This function uses a pre-defined tolerance while checking for coincident geometry; this tolerance can be changed using other functions in MergeTool.

After merging, the unmerged surfaces can be determined by counting the number of parent volumes. The print_unmerged_surfaces() demonstrates stepping through the global list of surfaces using the GeometryQueryTool::get_first_refface() and GeometryQueryTool::get_next_refface() functions (see Table 21). The CubitUtil::list_entity_ids() function takes a list of CubitEntity objects and prints the id numbers of those objects in a "pretty" format.

Table 21: Pseudo code for C++ driver; print_unmerged_surfaces() function.

```
CubitStatus print_unmerged_surfaces()
{
    // Print number and ids of non-shared surfaces
    // Non-shared surfaces are one with < 2 volumes connected to them,
    // or less than two parent entities
  RefFace *face = GeometryQueryTool::instance()->get_first_ref_face();

  DLIList<RefEntity*> parents;
  DLIList<RefFace*> unmerged_faces;

  for (i = 0; i < GeometryQueryTool::instance()->num_ref_faces(); i++) {

      // get the parent volumes
    parents.clean_out();
    face->get_parent_ref_entities(parents);
    if (parents.size() < 2) unmerged_faces.append(face);

    face = GeometryQueryTool::instance()->get_next_ref_face();
  }

    // now print information on unmerged surfaces

    // first cast faces to a cubit entity list; use cast_list_to_parent,
    // since it's much more efficient
  DLIList<CubitEntity*> temp_entities;
  CAST_LIST_TO_PARENT(unmerged_faces, temp_entities);
  PRINT_INFO("There were %d unmerged surfaces; their ids are:\n",
             unmerged_faces.size());
  CubitUtil::list_entity_ids("\0", temp_entities);

    // now we're done
  return CUBIT_SUCCESS;
}
```

3.8.6 Program output

The mergechk application was run using as input the geometry shown in Figure 1. The output of the driver program applied to this geometry is shown in Table 22. By default, the output from the calls to PRINT_INFO and PRINT_WARNING go to standard out; output from PRINT_ERROR goes to standard error. The definitions of these print macros are located in CubitMessage.hpp, and can be changed in cases where printed information is not desired. Alternatively, there are functions in CubitMessage which can be used to turn off printed output.

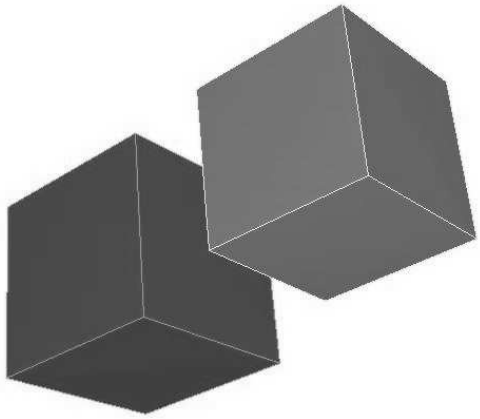

Figure 1: Sample geometry used to test mergechk application.

Table 22: Output of mergechk driver application.

```
=====================================
Read 2 ACIS Entities from the input file
Processing ACIS BODY 1, 2
Imported 2 Bodies: 1, 2
=====================================
ERROR: Intersection operation failed.
       Empty Body created.

Intersect finished;
Old bodies retained: 2
New bodies created:  (none)
=====================================
No body overlaps.
=====================================
10
Group imprint finished;
Old bodies retained:
New bodies created: 1 2
=====================================
Imprinting resulted in the following numbers of new entities:
   6 vertices.
   8 edges.
   2 faces.
=====================================

...Merging all features in the model

...Merging all Surfaces in the model
Surface 13 and 15 consolidated
Consolidated 1 surfaces

...Merging all Curves in the model
Consolidated 0 curves

...Merging all Vertices in the model
Consolidated 0 pairs of vertices
=====================================
There were 12 unmerged surfaces; their ids are:
2 to 7, 9 to 12, 14, 16
=====================================
```

3.8.7 Makefile

The makefile for the C++ mergechk application is shown in Table 23. The first thing to note about this makefile is that it defines a make variable for CGM_BASE_DIR. This variable defines the location of various parts of CGM, and should be set to one directory above the CGM libraries (i.e. the CGM libraries are located in CGM_BASE_DIR/geom. and …/util). Using this make variable, the makefile also includes the make.client file. This file defines many other make variables needed to compile CGM applications, therefore this file should be included in the makefile for all CGM applications. CUBIT_BASE_DIR should be defined to the same value as CGM_BASE_DIR; this variable is used for legacy reasons in CGM makefiles.

CGM application files are compiled using the default rules shown in Table 23. The CGM_INCLUDE variable tells the compiler where to look for include files which declare CGM functions. The other variables are defined in make.client, and provide other useful compiler flags. The ECHO_COMMAND and PREFIX variables are used to reduce the amount of screen output when applications are compiled; if the user wishes to see the actual command being used to compile CGM applications, they can simply remove PREFIX from the default compile rule or reset the value of the PREFIX variable in their makefile.

When linking CGM applications, it is very important to provide the CGM_LIBS_LINK and CUBIT_SYS_LINK variables as the last arguments to the link command before specification of the output file. This ensures that any CGM or ACIS functions used by applications will be resolved by the libraries listed in these variables. Including the CGM_LIBS variable on the list of dependents for the application will ensure that the application gets rebuilt when it is out of date with respect to the CGM libraries.

Table 23: Makefile for the mergechk CGM application.

```
CGM_BASE_DIR = /usr/local/eng_sci/cubit/src
CUBIT_BASE_DIR = /usr/local/eng_sci/cubit/src
include $(CUBIT_BASE_DIR)/util/OtherFiles/make.client

### main rule

mergechk: mergechk.o ${CGM_LIBS}
        ${LINKER} ${DEBUG_FLAG} ${LFLAGS} mergechk.o ${GEOM_LIBS_LINK} \
           ${EXTRA_GEOM_LIBS_LINK} ${GEOM_LIBS_LINK} ${CUBIT_SYS_LINK} -o mergechk

## replace some default suffix rules
.SUFFIXES : .o .cpp
.c.o :
        ${ECHO_COMMAND}
        ${PREFIX} ${CC} ${CFLAGS} ${MACH_CFLAGS} \
         ${CGM_INCLUDES} -c $<

.cpp.o :
        ${ECHO_COMMAND}
        ${PREFIX} ${CXX} ${CXXFLAGS} ${MACH_CXXFLAGS} \
         ${CGM_INCLUDES} -c $<
```

3.8.8 Summary

The driver application described in this section shows how CGM can be used to import, modify and query solid geometry. Initialization and termination of the package are simplified through simple API calls to singleton classes. A large variety of functions can be accessed through this interface, without needing to know the details of how those functions are implemented.

This same driver program has also been implemented in C, to demonstrate the use of C-language API functions written for CGM. The C-language version of mergechk is of comparable complexity to the C++-language version. Using the C interface, programs written in C or FORTRAN can also access and use CGM for solid geometry evaluation.

3.9 Facet Driver Application

In the previous section, a simple driver application was described, showing how to use CGM to access and modify solid geometry. In many cases, however, applications need to use facet-based geometry, and furthermore, would like to avoid the cost of linking solid modeling engine code. This section gives a simple example of how to do that.

The example described in this section shows how to initialize a facet-based model and query the resulting surfaces. Instructions are also given on how to build this application without linking the ACIS solid modeling engine. The pseudo-code below has many lines of the actual code removed, for brevity; the complete example is distributed with the CGM libraries, in the cgm_apps/examples/facetdriver subdirectory.

3.9.1 Forward declarations and main

The source code for forward declarations and the main function are shown in Table 17. This file begins with the inclusion of files containing declarations of functions and datastructures. In most CGM applications, GeometryQueryTool.hpp and/or GeometryModifyTool.hpp will be included, since these are the primary points of access to CGM. The FacetQueryEngine.hpp file is also included, for declaration of the facet geometry construction functions. In the main function, the first few lines of code initialize first CGM (by calling CGMApp::instance()->startup()), then the Facet and Geometry query tools. These classes are used to create and query facet-based geometry, respectively.

After CGM has been initialized and the respective geometry engine classes created, we are ready to begin creating facet-based geometry.

Table 24: Pseudo code for facet driver code, forward declarations and main.

```
// include tool and datastructure declarations
#include “GeometryQueryTool.hpp”
#include “FacetQueryEngine.hpp”
(…)

/// main program - initialize, then send to proper function
int main (int argc, char **argv)
{

    // initialize CGM
  CGMApp::instance()->startup(argc, argv);

    // Initialize the GeometryQueryTool and facet engine
  FacetModifyEngine *fqe = FacetModifyEngine::instance();
  GeometryQueryTool *gqt = GeometryQueryTool::instance();
```

3.9.2 Creating point and facet objects

CGM provides point and facet classes for holding data for those objects; the class hierarchy and design is described elsewhere In brief, there are abstract base classes CubitPoint and CubitFacet which define functions for accessing data, and leaf classes for actually storing those data. For applications which need CGM to store the facet data, the CubitPointData and CubitFacetData leaf classes should be used.

In this example, a unit cube is created, with two triangular facets per cube face. Table 25 shows the code for creating the point and facet objects for this geometry. The CPD and CFD typedefs are used as shorthand for the CubitPointData and CubitFacetData classes, respectively. Points are created for the eight corners of the cube (some lines have been left out for brevity; the unabridged source code for these examples can be found in an Appendix of this report). All points are put on a list, which is used later as input to the functions which make the faceted surfaces. Facets are created from successive point triples, corresponding to the facets bounding the unit cube. These facets are also put in a list. These lists are over

the abstract base class objects CubitPoint and CubitFacet, to allow applications to use the same facet-based geometry creation functions with other types of point and facet classes.

Applications deriving point and facet data from other sources would simply derive their own versions of CubitPoint and CubitFacet leaf classes, and build lists of these objects using those data. Those lists would be used as described below to create facet-based geometry.

Table 25: Pseudo code for facet driver, creating points and facets.

```
DLIList<CubitFacet*> f_list; DLIList <CubitPoint*> p_list;

  // define some really simple facets, corresponding to the
  // facets for a brick
typedef CubitPointData CPD; typedef CubitFacetData CFD;

CPD *p1 = new CPD(0, 0, 0); CPD *p2 = new CPD(1, 0, 0);
(…)
p_list.append(p1); p_list.append(p2);
(…)
CFD *f1 = new CFD(p1,p3,p2); CFD *f2 = new CFD(p1,p4,p3); // xy-
(…)
f_list.append(f1); f_list.append(f2); f_list.append(f3);
(…)
```

3.9.3 Creating facet-based geometry

Table 26 shows the code for creating the faceted surfaces and the resulting body for this example.

The points and triangular facets bounding the geometry we would like to construct are stored in point and facet lists, respectively. There are several options for creating surfaces from these facets. The surface(s) can be piecewise planar, corresponding exactly to the (planar) facets used as input; or, a smooth approximated surface can be fit over these points (see [2] for a description of how this is done). This option is controlled by an integer **`surf_type`** in Table 26. There is also an option for creating a single surface from these facets, or multiple surfaces separated based on a prescribed angle between neighboring facets. This angle is represented in Table 26 by the **`angle`** variable; using a value of 30 degrees ensures that for most models a single surface will be created. Using these parameters, the faceted surface(s) are created by calling the make_facet_surface function in FacetQueryEngine.

Once one or more faceted surfaces are created, those surfaces can be used as-is for normal and point evaluations, or they can be used to construct a body. The latter might be useful is a volume mesh of the region bounded by the surfaces is desired. Table 26 shows the steps necessary for creating a body from a collection of surfaces. The result is a topological body, pointed to by body_ptr, which is bounded by a collection of facet-based surfaces of our own definition.

Table 26: Pseudo code for facet driver; creating facet-based surface, shell, lump and body.

```
double angle = 30.0;
DLIList<Surface*> surf_list;
CubitStatus result;
result = fme->make_facet_surface(NULL, f_list, p_list, angle, 4,
                                 false, false, surf_list);
ShellSM *shell_ptr;
result = fme->make_facet_shell(surf_list, shell_ptr);
DLIList<ShellSM*> shell_list;
shell_list.append(shell_ptr);
Lump *lump_ptr;
result = fme->make_facet_lump(shell_list, lump_ptr);
DLIList<Lump*> lump_list;
lump_list.append(lump_ptr);

BodySM *bodysm_ptr;
```

```
Body *body_ptr;
result = fme->make_facet_body(lump_list, bodysm_ptr);

body_ptr = GeometryQueryTool::instance()->make_Body(bodysm_ptr);
```

3.9.4 Evaluation of topology and geometry

Once the geometry and topology have been created, applications determine what kind of evaluations of that geometry are desired. Table 27 shows an example of querying the topology of the model, then printing information about some of those topological entities. In this example, the number and position of geometric vertices in the model are printed.

Table 27: Pseudo code for facet driver; print number and positions of geometric vertices.

```
   // print vertex positions
DLIList<RefVertex*> verts;
gqt->ref_vertices(verts);
int i;
for (i = verts.size(); i > 0; i--) {
  CubitVector coords = verts.get_and_step()->coordinates();
  PRINT_INFO("Vertex %d: %f, %f, %f.\n",
             8-i, coords.x(), coords.y(), coords.z());
}
```

3.9.5 Evaluating facet-based surfaces

Table 28 shows how facet-based surfaces are queried for closest point and normal. A pointer to the first geometric face is obtained from GeometryQueryTool, then the face is queried for a point closest to the coordinate triple (.5, .5, 0). The normal of the surface at this point is also queried. Note that the RefFace class defines many other functions for querying the geometry of a surface; for a complete list of these functions, see the RefFace.hpp header file.

In some cases, applications need to evaluate individual facets in the model, for example when using a smooth surface approximation. This is done using the barycentric coordinates on the triangular facets. The CubitFacet::evaluate() function is used for this purpose, as shown in Table 28.

Table 28: Pseudo code for facet driver; evaluate geometric surface and facet.

```
RefFace *face = gqt->get_first_ref_face();
  // find closest point to several places
CubitVector test_position, result_position, normal;

test_position.set(.5, .5, 0);
face->find_closest_point_trimmed(test_position, result_position);
normal = face->normal_at(result_position);
PRINT_INFO("Point (%4.2f, %4.2f, %4.2f):\n  "
           "closest=(%4.2f, %4.2f, %4.2f), normal=(%4.2f, %4.2f, %4.2f).\n",
           test_position.x(), test_position.y(), test_position.z(),
           result_position.x(), result_position.y(), result_position.z(),
           normal.x(), normal.y(), normal.z());

double a = 1.0/3.0, b = 1.0/3.0, c = 1.0/3.0;
CubitVector temp_point(a,b,c), eval_point;
f1->evaluate( temp_point, &eval_point );
PRINT_INFO("Evaluation of facet 1 at (%4.2f, %4.2f, %4.2f) is "
           "(%4.2f, %4.2f, %4.2f).\n",
            a, b, c, eval_point.x(), eval_point.y(), eval_point.z() );
```

3.9.6 Program output

The output of this example is shown in Table 29. As expected, one surface is created, with no bounding curves or vertices. When evaluated in the center of each face, the closest point is clearly on a curved surface, while the normals are parallel to the coordinate axes, as expected.

As expected, 6 surfaces are created, along with eight vertices at the corners of the cube. All surface evaluations fall on unit faces of the cube, and normals are parallel to the unit vectors, as expected.

Table 29: Output of facet driver application, angle = 30.

```
Constructed 1 surfaces.
Body successfully created.
Number of vertices = 0
Number of edges = 0
Number of faces = 1
Number of volumes = 1
Number of bodies = 1
Point (0.50, 0.50, 0.00):
  closest=(0.50, 0.50, -0.30), normal=(-0.07, 0.07, -1.00).
Point (0.50, 0.50, 1.00):
  closest=(0.50, 0.50, 1.30), normal=(0.07, 0.07, 1.00).
Point (0.50, 0.00, 0.50):
  closest=(0.50, -0.30, 0.50), normal=(0.07, -1.00, -0.07).
Point (0.50, 1.00, 0.50):
  closest=(0.50, 1.30, 0.50), normal=(0.07, 1.00, 0.07).
Point (0.00, 0.50, 0.50):
  closest=(-0.30, 0.50, 0.50), normal=(-1.00, -0.07, 0.07).
Point (1.00, 0.50, 0.50):
  closest=(1.30, 0.50, 0.50), normal=(1.00, -0.07, -0.07).
Evaluation of facet 1 at (0.33, 0.33, 0.33) is (0.73, 0.27, -0.25).
Evaluation of facet 2 at (0.00, 1.00, 0.00) is (0.00, 1.00, 0.00).
Evaluation of facet 3 at (0.50, 0.50, 0.00) is (0.50, -0.11, 1.11).
Evaluation of facet 4 at (0.00, 0.50, 0.50) is (0.50, 1.11, 1.11).
```

Table 30 shows the output of a modified facet driver application, where an angle of 135 degrees is used. With this input, any facets whose interior angle is less than 135 degrees will be separated by a topological curve, with a C1 discontinuity in the surface as well. Since the facets input form a logical cube, we expect to see that cube recovered in the geometric topology. Indeed, this is the result we get, i.e. there are eight vertices, twelve curves and six faces in the resulting model. Interestingly, the closest point evaluations all fall on planar surfaces; this is because, for each surface in the model, the facets making up each surface are co-planar.

Table 30: Output of facet driver application, angle = 135.

```
Constructed 6 surfaces.
Body successfully created.
Number of vertices = 8
Number of edges = 12
Number of faces = 6
Number of volumes = 1
Number of bodies = 1
Vertex 0: 1.00, 1.00, 1.00.
Vertex 1: 0.00, 1.00, 1.00.
Vertex 2: 0.00, 0.00, 1.00.
Vertex 3: 1.00, 0.00, 1.00.
Vertex 4: 1.00, 1.00, 0.00.
Vertex 5: 1.00, 0.00, 0.00.
Vertex 6: 0.00, 0.00, 0.00.
Vertex 7: 0.00, 1.00, 0.00.
Point (0.50, 0.50, 0.00):
  closest=(0.50, 0.50, 1.00), normal=(0.00, 0.00, 1.00).
Point (0.50, 0.50, 1.00):
  closest=(0.50, 0.50, 1.00), normal=(0.00, 0.00, 1.00).
Point (0.50, 0.00, 0.50):
  closest=(0.50, 0.00, 1.00), normal=(0.00, 0.00, 1.00).
Point (0.50, 1.00, 0.50):
```

```
  closest=(0.50, 1.00, 1.00), normal=(0.00, 0.00, 1.00).
Point (0.00, 0.50, 0.50):
  closest=(0.00, 0.50, 1.00), normal=(0.00, 0.00, 1.00).
Point (1.00, 0.50, 0.50):
  closest=(1.00, 0.50, 1.00), normal=(0.00, 0.00, 1.00).
Evaluation of facet 1 at (0.33, 0.33, 0.33) is (0.67, 0.33, 0.00).
Evaluation of facet 2 at (0.00, 1.00, 0.00) is (0.00, 1.00, 0.00).
Evaluation of facet 3 at (0.50, 0.50, 0.00) is (0.50, 0.00, 1.00).
Evaluation of facet 4 at (0.00, 0.50, 0.50) is (0.50, 1.00, 1.00).
```

3.9.7 Makefile

The makefile for the facets application is shown in Table 31. In most ways, this makefile is similar to that of the mergechk application in Table 23. The most important difference is in the link line, where the ${EXTRA_GEOM_LIBS_LINK} variable is *not* present, as it is in the mergechk makefile. This make variable is the one which contains definitions for the ACIS libraries; not including this variable in the facets link line results in facets being linked without the ACIS libraries. This saves a substantial amount of memory in the application.

Table 31: Makefile for the facets CGM application.

```
CGM_BASE_DIR = /usr/local/eng_sci/cubit/src
CUBIT_BASE_DIR = /usr/local/eng_sci/cubit/src
include $(CUBIT_BASE_DIR)/util/OtherFiles/make.client

### main rule

facets: facets.o ${GEOM_LIBS}
        $(MAKE) libs
        ${LINKER} ${LFLAGS} ${MACH_LFLAGS} ${DEBUG_FLAG} facets.o ${GEOM_LIBS_LINK} \
        ${CUBIT_SYS_LINK} -o facets

## replace some default suffix rules
.SUFFIXES : .o .cpp
.c.o :
        ${ECHO_COMMAND}
        ${PREFIX} ${CC} ${CFLAGS} ${MACH_CFLAGS} \
         ${CGM_INCLUDES} -c $<

.cpp.o :
        ${ECHO_COMMAND}
        ${PREFIX} ${CXX} ${CXXFLAGS} ${MACH_CXXFLAGS} \
         ${CGM_INCLUDES} -c $<
```

3.9.8 Summary

The facets application described in this section shows how CGM can be used to create and query facet-based geometry. While this example was done using hard-wired point positions and facet topologies, it would be simple to change to account for run-time-defined facets.

4 Application Developer's Guide

4.1 Introduction

There are two types of applications of CGM; the first type is where an application relies entirely on CGM for its geometry functionality, and does not require addition of data to CGM geometry objects or extension of their functional capabilities. Another use of CGM, and one which is much more powerful, is to use CGM geometry objects as foundation classes, upon which additional functionality is implemented. For example, classes can be derived from the CGM basic geometry entity types (RefVertex, RefEdge, etc.)

which not only provide geometry functionality, but which also (for example) provide mesh-related functions and datastructure.

CGM has been carefully designed to support applications of the latter type. In fact, the CUBIT application is built on CGM by deriving CUBIT-specific geometry classes, which enrich the CGM geometry classes with functions for generating and storing mesh. Since class derivation is used to extend the CGM classes, these extended classes must be created and used in place of the unenriched CGM classes, even if that creation is taking place from within CGM. This is accomplished using Factory classes and other advanced designed techniques; these techniques are described in this section.

This section begins by describing the CGM geometry entities which can be enriched using application-specific subclasses. The Factory class used to create geometry entities is then described, along with how this class can be extended to create application-specific geometry entities. Sections 3.7.5 and 4.3 describe geometry attributes, which are used to associate application-specific data to geometry entities. Section 4.4 describes the ToolData mechanism, which can be used to associate application-specific data to many types of data including geometry entities. Section 4.5 describes the CubitObserver mechanism, which is used to notify application-specific objects of changes to geometry entities and other observable objects. This chapter concludes with specific information on how CUBIT uses these mechanisms to implement meshing functionality on top of the CGM classes.

4.2 RefEntityFactory: Topological Entity Construction

As described earlier, the children of the RefEntity class are RefVertex, RefEdge, RefFace, RefVolume, Body, RefGroup, RefPart, RefAssembly, and RefCoordSys. These entities are most useful for implementing applications using CGM, and are therefore the entities from which child classes can be derived. In the future, derivation of classes from other entities, like Loops and CoEdges, may be allowed.

Also described earlier was the relationship between entities in the solid model file and the corresponding CGM entities; for example, each FACE in an ACIS model has a corresponding RefFace in the CGM model. When geometry is imported or created, the CGM entities are constructed for each entity in the solid model; this construction is implemented in CGM code, in particular in GeometryQueryTool. However, if an application derives child classes from the CGM entities, the application-specific objects must be constructed instead. Constructors for the application-specific child classes cannot be called from the CGM code, since this would make CGM dependent on those applications. Therefore, the construction of RefEntity's in the geometric model is implemented in a singleton factory; CGM is designed to allow this factory to be replaced with an application-specific factory, which constructs application-specific derived class objects and returns them as CGM objects.

Consider first the CGM factory, implemented in the RefEntityFactory class. Following the singleton pattern, there is only a single RefEntityFactory object created for the application; this object is created the first time RefEntityFactory::instance() is called. Afterwards, this function simply returns a pointer to the singleton factory instance (a static pointer to the instance is maintained in RefEntityFactory). CGM requires that all RefEntity's be constructed using RefEntityFactory (this is accomplished by making the constructors of RefEntity leaf classes inaccessible from other classes). RefEnityFactory declares virtual functions for constructing geometric entities, e.g. RefEntityFactory::construct_RefFace, RefEntityFactory::construct_RefVolume, etc.

To substitute an application-specific factory for RefEntityFactory, an application simply derives that factory from RefEntityFactory, and constructs that application-specific factory before RefEntityFactory::instance() is called (this is typically done just before initializing GeometryModifyTool). A pointer to the application-specific factory is still stored in RefEntityFactory, and returned from RefEntityFactory::instance(). However, since RefEntityFactory::construct_xxx functions were declared virtual, the application-specific factory can substitute alternative implementations for these functions; these implementations simply construct application-specific entities, passing them back as the parent class pointer.

For example, say an application wants to extend the functionality of vertices only. The application would write a derived entity factory, ARefEntityFactory:

```
Class ARefEntityFactory : public RefEntityFactory
    {
public:
    ARefEntityFactory *instance()
        {
    if (instance_ == NULL) instance_ = new ARefEntityFactory;
    return instance_;
        }

Virtual RefVertex *construct_RefVertex(…);
    }
```

Any CGM code calling RefEntityFactory::construct_RefVertex would actually call the function in ARefEntityFactory, thanks to the virtual function mechanism in C++. This way, an application-specific vertex is created instead of an unextended vertex object. Note that, since the extended vertex is derived from RefVertex, all functions defined for RefVertex and its parent classes can be called directly from ARefVertex. Therefore, the ARefVertex objects can be used for normal CGM-type topology traversal, as well as for application-specific uses.

Details on how CUBIT uses this mechanism to implement meshing functionality on geometry objects are given in the next chapter.

4.3 Application-Specific Attributes

Geometry attributes are used to store application-specific data directly on objects to which they are associated. This data is saved to and restored from the solid model files automatically. Geometry attributes used by CGM include entity names, ids, group membership, and others. CGM also provides a means for applications to define their own attributes, which are then managed by CGM along with and in the same way as CGM attributes.

Similar to how application-specific RefEntity objects are managed, CGM utilizes a factory pattern for the creation of attributes. A pure virtual base class, named CubitAttribFactory, is defined with two pure virtual functions, both named create_cubit_attrib; these functions create an application-specific attribute from an attribute type and a simple attribute pointer, respectively.

The process for defining an application-specific attribute factory is as follows. First, the application-specific attribute classes are designed and written (see other attributes like CAEntityName and CAEntityId for guidelines on writing attributes). Then, an application-specific attribute factory is named and written (in CUBIT, this class is named CAFactory); this factory is derived from CubitAttribFactory:

```
Class AppCAFactory : public CubitAttribFactory
    {…}
```

Two functions need to be declared and defined:

```
CubitAttrib *AppCAFactory::create_cubit_attrib(const int attrib_type,
RefEntity *owner)
    {…}

CubitAttrib *AppCAFactory::create_cubit_attrib(CubitSimpleAttrib *csa_ptr,
RefEntity *owner)
    {…}
```

Both these functions must be defined, since they are called from CubitAttrib.

The constructor for CubitAttribFactory is declared to be protected, which means only child and friend classes of CubitAttribFactory can construct this class. Typically, an application-defined attribute factory will be written with a static member which constructs the factory. Inside that function, the factory must be constructed and then passed to CubitAttrib, which stores a pointer to it:

```
static AppCAFactory *create_factory() {
```

```
ApplicationCAFactory *factory = new ApplicationCAFactory();
CubitAttrib::set_cubit_attrib_factory(factory);
    }
```

This factory should be created before the application imports any solid model files, otherwise any application-specific attributes in those files will not be initialized automatically.

There are several static variables and one field in an enum corresponding to each type of attribute. These variables are described in Table 32. Currently, these variables and the enum are defined in CubitAttrib.hpp, inside the CGM code. This requires changes to CGM if applications choose to add more attributes. This will be changed eventually to allow the definition of these variables in the application-specific attribute factory.

Table 32: Variables and enum defined for each attribute. Currently, these variables are defined in CubitAttrib.hpp.

Variable / Enum	Description
`AutoActuateFlag[t], autoUpdateFlag[t]`	When true, attributes of type t actuate/update automatically after restoration / before saving from/to solid model file.
`AutoReadFlag[t], AutoWriteFlag[t]`	When true, attributes of type t are read from/written to the solid model file automatically.
`ActuateInConstructor[t]`	When true, attributes of type t are actuated from a call inside the object constructor; otherwise, they are attributed after all entities have been read from the solid model file in which the attributes resided.
`ActuateAfterGeomChanges[t]`	If true, attributes of type t are not actuated until after any changes to the geometry resulting from other attributes have taken place.
`CubitAttributeType`	Enum of attribute types; new attributes should be defined before CA_LAST_CA in CubitAttrib.hpp.
`AttTypeName[t]`	User-visible attribute type name of attribute t.
`AttInternalName[t]`	Internal name for attribute type t (written to solid model file for identification purposes).

When an attribute's AutoActuateFlag flag is set to true, attributes of that type are actuated automatically when the solid model file is imported. If the attribute's ActuateInConstructor flag is set to true, the attribute is actuated from inside the constructor of the geometry entity to which it is associated, otherwise it is actuated after the entire solid model is imported and the corresponding geometry entities created. Actuation of application-specific attributes should be done carefully, especially if those attributes rely on functions or data stored on application-specific geometric entities derived from the RefEntity leaf classes (RefVertex, RefEdge, etc.). In this situation, the actuate() function for these attributes will be called from the RefEntity leaf class constructor, before the application-specific entity has been created, so casting to an application-specific object will fail. Application-specific attribute actuate functions should check to make sure they are applied to the correct type of entity, and if not, should return CUBIT_FAILURE. The actuate function can be called again from the application-specific entity constructor by calling CubitAttribUser::auto_actuate_cubit_attrib(), at which time the attribute can actuate correctly.

4.4 ToolData: Application-Specific Data

There are times when applications need to store data on geometric entities that is of a transient nature. For example, the mapping algorithm in CUBIT stores angle information on the vertices bounding surfaces and volumes. This data is no longer used after the meshing algorithm is finished generating mesh, and so there is no need to store it in the datastructure of the geometric entity. CGM provides a mechanism for storing transient data on entities; this mechanism is referred to as the ToolData capability.

This capability consists of two base classes, ToolData and ToolDataUser. ToolDataUser is a base class of RefEntity, and is used to manage a list of ToolData objects. ToolDataUser implements functions for

adding, removing, and returning a list of ToolData objects. Classes derived from ToolData are used to store the transient information which should be associated with the ToolDataUser. ToolData also stores a pointer to the next ToolData in a list. Thus, ToolData objects are stored in a singly linked list, the head of which is pointed to by ToolDataUser.

Implementing new types of ToolData objects is quite simple. ToolData-derived classes are typically named TDSomeName, and are required to implement an identification function:

```
int TDSomeName::is_some_name()
{return (get_address(TDSomeName_TYPE) ? CUBIT_TRUE : CUBIT_FALSE);}
```

This function is called from ToolDataUser to identify ToolData types:

```
TDSomeName my_td =
(TDSomeName *) tool_data_user->get_TD(&ToolData::is_some_name);
```

See documentation on ToolData, ToolDataUser, and some of the CGM ToolData classes (TDCompare, TDUniqueId) for more details.

4.5 CubitObserver: Application-Specific Observation of Entities

One of the classic problems in object-oriented design is how to allow one class (the observer) "observe" another class (the observable), without making the implementation of observable know about the implementation of the observer. This problem is addressed by using the well-known Observer pattern in C++. CGM's implementation of observers, in CubitObserver and CubitObservable, is modeled directly after that pattern.

To implement an application-specific observer, the developer needs to do the following:

- Derive the application-specific class, e.g. ASObserver, from CubitObserver
- Implement ASObserver::notify_observer(CubitObservable *observable, EventType observer_event, CubitBoolean from_observable), which handles events generated by observables.
- Use the functions CubitObserver::register_observable and CubitObserver::unregister_observable to register and unregister the application-specific observer with the observable (CubitObserver::unregister_observable does not necessarily need to be called, since it is called in the CubitObserver destructor).

The EventType enum is defined in CubitDefines.h.

There is also a mechanism in CubitObserver for implementing "static" observers, that is observers which observe all events. Static observers can be registered by calling CubitObserver::register_static_observer and CubitObserver::unregister_static_observer. This mechanism is used in CUBIT to implement some graphics functions and to manage some global entity lists, for example.

Applications are also free to implement their own observables as well as observers to observe them. In fact, there are some classes in CGM which are derived from both CubitObservable and CubitObserver. For example, RefGroup is both a CubitObserver, to keep track of entities which are in the group but which should get removed upon deletion, as well as a CubitObservable, since it too can be contained in groups.

4.6 Example: CUBIT Implementation Using CGM

In this example, details are given about how CUBIT is implemented on top of CGM. This purpose of this example is to show how an application might extend the capabilities of CGM by deriving classes from the CGM classes. It also serves as an introduction to CUBIT developers of the CUBIT database.

4.6.1 Derived topological entities

In addition to representing geometry, geometric entities in CUBIT must also be able to store mesh, represent finite element boundary conditions, and serve as a point of access to meshing algorithms assigned to those entities. This is done using inherited classes.

The inheritance hierarchy used in CUBIT is shown in . As the figure shows, from each RefXxx leaf class in CGM (RefVertex, RefEdge, RefFace, RefVolume, Body and RefGroup), CUBIT derives a corresponding MRefXxx class (MRefVertex, MRefEdge, etc.). CUBIT also defines a parent class for these leaf classes, which is named MRefEntity. MRefEntity serves as a common base class for meshable entities in CUBIT, and is decorated by (i.e. derived from) various classes which define certain meshing-related functions. The parent classes of MRefEntity are MeshContainer, used to store mesh on an MRefEntity, and MeshToolUser, used to associate meshing tool data with the entity, including the meshing scheme assigned to the entity.

4.6.2 MRefEntity class

Using a common MRefEntity base class simplifies the design of the MRefXxx classes by allowing the implementation of meshing functionality at a higher level in the derivation hierarchy. However, it also complicates things because of the use of multiple inheritance. Much of the parsing code in CUBIT operates on lists of MRefEntity's, since the parsing involves setting meshing parameters, but it also needs to access functions defined above RefEntity, list CubitEntity::class_name and CubitEntity::id. For these reasons, most of the commonly-used functions in RefEntity and above are defined as virtual functions, and can be overridden in derived classes. Using this technique, common functions like class_name and id are defined as virtual functions in MRefEntity as well, so that they can be used with MRefEntity objects. These functions are defined in the MRefXxx leaf classes, and call the CubitEntity functions explicitly.

CGM applications not needing a common base class for derived geometric entities need not implement one. If an application does define a common base class like that, but does not implement those common functions, those functions are only accessible from objects of the leaf classes or from CGM objects.

4.6.3 Topology traversal functions

One of the most common operations on geometric entities is to access topologically connected entities, for example finding all the edges containing a given vertex. These functions are implemented in TopologyEntity, and can be called directly from RefXxx objects. However, these functions pass back lists of RefXxx objects, which may still need to be casted to MRefXxx objects. To solve this problem, a similar set of topology traversal functions is implemented in MRefEntity which return lists of MRefXxx objects; these functions have the same name as those in TopologyEntity, except with an 'm' prepended to their names, e.g. mref_edges, mref_faces, etc. To access the ref_xxx functions directly from an MRefEntity object, the MRefEntity::topology_entity() function should be used to first cast the entity to a TopologyEntity, on which the ref_xxx() function can be called directly.

4.6.4 MRefEntityFactory

As described in the previous chapter, a factory class is used to create geometric entities in CGM. This allows the definition of application-specific geometric entities, which are created from an application-specific factory. CUBIT defines the MRefEntity class for this purpose. Note that this factory class must be instantiated before the creation of any geometry in the application; in CUBIT, this is done by calling MRefEntityFactory::instance() just before the first call to GeometryModifyTool::instance().

If an application-specific factory is implemented, it is also responsible for the storage of global lists of geometric entities; global list functions like ref_volumes(), ref_faces(), etc. must also be defined (these functions are declared virtual in RefEntityFactory for this purpose). CUBIT implements this by defining pointers to global lists in RefEntityFactory. These lists are created in the first call to RefEntityFactory::instance(), just after calling the RefEntityFactory constructor; if an application-specific entity factory has already been created, this part of RefEntityFactory::instance() is never executed, and the list pointers remain NULL.

4.6.5 AttribFactory for CUBIT-specific attributes

As described in the previous chapter, applications can define their own attributes by providing an application-specific attribute factory. This factory should be instantiated before any attributes are created.

In CUBIT, the CAFactory serves this purpose, and is instantiated before the first call to GeometryModifyTool::instance(). CAFactory is derived from CubitAttribFactory, which is itself declared in the CubitAttrib class header.

Currently, CUBIT-specific attribute types are part of the CubitAttributeType enum, defined in CGM. Technically, this should not be done, since it requires modification of CGM to add application-specific attributes. Other applications can easily bypass this by defining their own use of the CubitAttributeType values above the ones used in CGM (those currently end at CA_DEFERRED_ATTRIB in CubitAttrib.hpp). This will be changed eventually to allow the definition of arbitrary attribute types (identified by an integer).

4.6.6 DrawingTool

The DrawingTool class in CUBIT is used to implement graphics functionality for the entire code. As such, this class depends on virtually all of the datastructure classes in CUBIT, and many of the datastructure classes depend on DrawingTool. Eventually, this backward dependence will be removed and replaced entirely by the static observer mechanism described earlier. That implementation will be done incrementally. In the meantime, the GdrawingTool class in CGM serves as a parent class of DrawingTool, and simply provides empty functions for those DrawingTool functions which are called from within CGM. Applications are welcome to derive their own drawing mechanism from GDrawingTool (see GDrawingTool for more details).

4.6.7 The Model class

It has proven useful in CUBIT to have a single point of access to all the geometry, mesh and boundary condition data created during the meshing process; the Model class serves that purpose in CUBIT. While the actual data may not be stored there, Model acts as a common point of reference. The functions in Model then call the appropriate functions in MeshContainer, CGM, or wherever else to retrieve the necessary data. Other applications can easily port to using CGM by retaining their equivalent of the Model class, and modifying it to provide access to CGM functions.

4.6.8 Adding mesh data to MRefGroup

In addition to adding various types of decoration to the basic geometry capabilities in the RefEntity classes, sometimes there is a need to modify the fundamental purpose of one of those datastructures. In the CUBIT application, RefGroup has been modified to store not only geometry entities but also mesh entities. This required the addition of functions necessary for adding, removing and accessing the mesh stored in a group. MRefGroup is still derived from RefGroup (as well as MRefEntity), but also extends the functional interface of RefGroup by defining MRefGroup functions and datastructure. Thus, applications can use a combination of class decoration and class extension to add functionality to the basic CGM geometric entities.

5 CUBIT Usage of CGM

5.1 MRefEntity Class Structure

The structure of the MRefEntity class is shown in . Note that MRefEntity is not derived from RefEntity. While virtual inheritance could have been used to derive MRefEntity from RefEntity, this was not done for reasons of efficiency, and because of the already complicated inheritance hierarchy below the TopologyEntity and RefEntity classes in CGM. The MRefXxx leaf classes derive from both MRefEntity and their RefXxx counterparts.

Since the parsing code works extensively with lists of MRefEntity objects, many of the functions defined in RefEntity, TopologyEntity and CubitEntity have been re-defined in MRefEntity. In order to avoid ambiguities when calling those member functions on MRefXxx objects, definitions in the MRefXxx classes

were needed as well. Further complicating this problem, some of those functions were already re-defined in some of the RefXxx classes (e.g. bounding_box()). To simplify definition of these functions, macros were written and are stored in MRefFuncs.hpp. Four macros are defined:

- MREFFUNCS_DECLARE(CLASS)
  Declares functions common to MRefEntity and MRefXxx classes; CLASS argument is used in the (inline) definition of the entity_type() function.
- MREFFUNCS_DIFFERENT_IMPLEMENTATIONS
  This set of functions has a common declaration across MRefEntity and its children, but the definitions are different (e.g. bounding_box() considers only the entity for MRefVertex-MRefBody, but considers all the contained entities for MRefGroup). #include'd in the MREFFUNCS_DECLARE macro.
- MREFFUNCS_DEFINE1(CLASS)
  Common definitions of functions declared in MREFFUNCS_DECLARE; #include'd in the MRefXxx.cpp files. CLASS argument is used to qualify the function definition, e.g. CLASS::class_name().
- • MREFFUNCS_DEFINE2(CLASS)
  Common definitions of functions declared in MREFFUNCS_DECLARE; #include'd in the MRefEntity.cpp files. CLASS argument is used to qualify the function definition, e.g. CLASS::class_name(). Definitions differ from those in MREFFUNCS_DEFINE1 because they call the functions through the return value of ref_entity() instead of calling them directly (this is done because MRefEntity is not a direct descendent of RefEntity).

In addition to including these macros in the MRefXxx.hpp and MRefXxx.cpp files, definitions are added for the functions in the MREFFUNCS_DIFFERENT_IMPLEMENTATIONS macro.

Care should be taken when adding a function to those declared/defined in these macros, to make sure that both the definitions are added (to MREFFUNCS_DEFINE1 and MREFFUNCS_DEFINE2) as well as the declaration to MREFFUNCS_DECLARE. It should be relatively rare that functions are added to these macros at all.

When debugging these functions, the debugger will not be able to step into the functions defined in these macros. However, if the "step into" debugger function is used, the debugger will step into the definition of the function in CGM, i.e. in RefEntity, CubitEntity, or wherever else it is defined.

5.2 Library Initialization

Because of the extensibility features built into CGM, and the extension of those classes by CUBIT, the initialization of CGM is a bit more complicated for CUBIT. In particular, CUBIT must initialize MRefEntityFactory and CAFactory before creating or using GeometryModifyTool. This is most easily done by calling MRefEntityFactory::instance() and CAFactory::create_factory before calling GeometryModifyTool::instance() for the first time. Also, there are several tools which use mesh-defined geometry, in particular CompositeToolMesh and PartitionToolMesh; these classes are derived from the virtual geometry classes CompositeTool and PartitionTool, respectively, and also must be initialized before creating any virtual geometry. The initialization of these tools is done in main(), before calling GeometryModifyTool::instance().

The mechanics of how these CUBIT-specific tools are stored and used inside CGM varies a bit. MRefEntity, CompositeToolMesh and PartitionToolMesh are all derived from their respective parents who are singleton classes. Singleton classes store a static pointer to the (single) instance of that class; the child classes simply create a child object and store a pointer to the child in the parent's static pointer. Thus, when the parent class's instance() function is used, it returns the pointer to the child class. CAFactory, and other classes derived from CubitAttribFactory, work slightly differently. CubitAttrib keeps a static pointer to a CubitAttribFactory object; this pointer is initialized to NULL, but gets assigned inside CAFactory::create_factory to point to the CAFactory object. This object is used from CubitAttrib if attributes unknown to CubitAttrib are encountered.

5.3 Access To Geometric Model

5.3.1 Global lists

As stated earlier, pointers to global lists of entities are maintained in both RefEntityFactory and MRefEntityFactory. Only one set of those pointers is non-NULL; in the case of CUBIT, the lists are non-NULL in MRefEntityFactory. Thus, global lists are stored and accessed from MRefEntityFactory; the functions for accessing global lists in RefEntityFactory (e.g. ref_faces(), ref_volumes()) are declared virtual, and are overridden in MRefEntityFactory.

Although the global list access functions are declared public in MRefEntityFactory, in most cases these lists should be accessed through either Model or GeometryModifyTool. In fact, for most of the code in CUBIT (i.e. the code outside CGM), these lists should be accessed through Model. Model provides a common access point for most of the datastructure in CUBIT (geometry, mesh and boundary conditions). The functions in Model are declared with the same names as those in MRefEntityFactory, to minimize confusion. Functions are provided for returning list copies (mref_faces, etc.), for returning a single entity of a given type (mref_face(), etc.), or for direct (but read-only) manipulation of the lists (get_first_mref_face(), get_next_mref_face(), get_last_mref_face(), etc.). Similar functions are declared in GeometryQueryTool, for accessing global lists of RefXxx entities.

In rare cases, efficiency concerns may be of such concern that direct access to the global lists is necessary. There are functions defined in MRefEntityFactory which return the pointers to the global entity lists, which can be used to modify the lists directly. These functions should only be used when the functions in Model or GeometryQueryTool cannot be used, for efficiency or other reasons. See the documentation for MRefEntityFactory for a description of these functions.

5.3.2 Topological queries

One of the most common types of geometry query in CUBIT is topological queries, e.g. return a list of edges bounding a face. Since lists of entities do not derive from one another based on the inheritance hierarchy of the entities in the list, we need functions which return lists of MRefXxx objects that are distinct from the corresponding topological query functions defined in TopologyEntity. That is, we need functions like mref_edges(), mref_faces(), etc. that return lists of MRefEdge's and MRefFace's, respectively. These functions are defined in MRefEntity, and so can be used with any object derived from MRefEntity.

5.4 MRefEntity Construction/Destruction

Because of the extensibility features of CGM and the use of factories to construct geometric entities, the implementation of the MRefEntity constructors and destructors can be rather complicated. This section describes these implementations in more detail.

5.4.1 Construction

The constructors of the RefXxx classes are all private. This is to prevent any code from constructing RefXxx objects without going through the RefEntityFactory. This ensures that any code in CGM requesting a new RefXxx object will actually get an application-specific RefXxx object (e.g. MRefXxx object in CUBIT) in return. Constructors in the MRefXxx classes are private for the same reason. Because all these constructors are private, the RefXxx and MRefXxx classes declare RefEntityFactory and MRefEntityFactory, respectively, as friend classes; this allows the factory classes to call those private constructors.

Tools needing to construct new geometric entities must therefore call functions in the appropriate factory class. For example, RefEntityFactory::construct_RefEdge(Curve *) constructs and returns a RefEdge given

a Curve pointer, while MRefEntityFactory::construct_MRefEdge(Curve *) constructs an MRefEdge and returns a pointer to an MRefEdge.

```
GeometryTool::delete_Body(Body *thisbody):

- delete solid model entities on thisbody and all lower order geometry
- call thisbody->ModelEntity::remove_from_DAG():
  - for all children (lower order geometry):
    - remove thisbody from child's parent list
    - call child->remove_from_DAG()
    - if successful, add child to deactivated list
    - notify observers of child that child is deleted
  - (end for)
  - add thisbody to deactivated list
  - notify thisbody observers that thisbody is deleted
  - call GeometryTool::cleanout_deactivated_geometry()
- (end ModelEntity::remove_from_DAG())

GeometryTool::cleanout_deactivated_geometry():

- call DAG::cleanout_deactivated_DAG_nodes():
  - for each deactivated DAG node:
    - delete deactivated DAG node; in DAG node destructor:
      - set CDO backpointer to DAG node to NULL
      - delete CDO; in CDO destructors (CDO is a ModelEntity, which is a parent
        of all the geometric entities):
        - if CDODAGNodePtr is not NULL, set to NULL and delete
                    (this should never happen!)
      - (end CDO destructors)
    - (end DAG node destructor)
```

Figure 2: Pseudo code for deleting geometric entities.

5.4.2 Destruction

Destruction of geometric entities in CUBIT was quite complicated before CGM, and remains so. At the highest level, the GeometryQueryTool::delete_Body(Body *) function should be used to delete bodies. This function implements the logic necessary to delete the body, the solid model entities represented by the body, and any mesh that may exist on the body that isn't shared by geometric entities not being deleted.

Unfortunately, it is often necessary to understand what is happening during the entity deletion process. Pseudo code for this process, as implemented when this manual was written, is shown in Figure 2. The most important thing to derive from this information is that the destructors for the geometric entities are called directly from CDODAGNODE::~CDODAGNODE().

5.4.3 Construction/Destruction and Multiple Inheritance

By definition, when a class object inherits from multiple base classes, the base class constructors are called before that of the child class, in the order in which they appear in the child class declaration. Destructors are called in reverse order. As implemented in CUBIT, the MRefXxx classes inherit from RefXxx first, then from MRefEntity. This has important implications on what happens in the MRefXxx and MRefEnity constructors and destructors.

The primary thing to remember about constructors is that during the construction of an MRefXxx object, inside the MRefEntity constructor, the object does not yet know that it is also a RefEntity. This is because the leaf class, MRefXxx, which is the link between the two sets of parent classes, has not yet been constructed. So, if inside the MRefEntity constructor, an attempt is made to call ref_entity(), e.g. for the

purpose of traversing the topology, that function will return NULL. Thus, any work that needs to be done in the constructor that requires topology traversal should be done in the leaf classes, not in MRefEntity.

Likewise, during the destruction process, the leaf class MRefXxx is destroyed before calling the MRefEntity destructor. Any call to ref_entity() inside the MRefEntity destructor will also return NULL. Therefore, any destruction-related code which requires topology traversal should be called from the MRefXxx destructors. For example, this requires that CubitObservable::remove_from_observers() be called from the MRefXxx destructors, since observers sometimes perform topology traversal from the entity they are un-observing.

5.5 Observer Notification

In order to remove dependency of CGM on CUBIT classes like DrawingTool and Model, while still being able to notify those classes of changes to the geometry, a general observer mechanism was implemented. Using this mechanism, objects derived from CubitObserver can "observe" objects derived from CubitObservable, and can be notified when these objects are modified or destroyed. The observer mechanism works by keeping, for every observable, a list of observers which get notified upon changes to the observable. Although this same mechanism could be used to implement the observation of entities by DrawingTool and Model, this would be inefficient, since most mesh and geometry entities would keep pointers to these classes. Instead, a static observer mechanism was also implemented, where certain classes can register themselves to observe all events from all observables.

5.5.1 Static observers

Static observers register themselves by calling CubitObserver::register_static_observer(CubitObserver*). A pointer to each static observer is kept on a static observer list in CubitObserver (actually, CubitObserver just points to that list, creating it from CubitApp::instance() upon startup). Typically, static observers are also singleton classes, which means they can be registered inside either the constructor or the part of the instance() function which calls the constructor (the latter method is used when there may be child classes derived from those singleton classes, to avoid the same object being registered twice). The current static observers in CUBIT are DrawingTool and Model.

Static observers are notified of events by calling CubitObserver::notify_static_observers(EventType).

5.5.2 Construction

When entities are constructed, several things are done at the global level. The entity must be added to global lists, and if graphics are active, the entity is added to the graphics display list. In CUBIT, there are specific events which accomplish these and other things; these events are listed in Table 33.

Note that notify_static_observers(MODEL_ENTITY_CONSTRUCTED) is called from both the MRefXxx and RefXxx constructors. The implementation of this notify function is done carefully, to avoid adding a given entity to the global lists twice. This is accomplished in the following way. First, inside the Model::notify_observer function, the CubitObservable argument is cast to an MRefEntity; if that cast is successful, the Model::notify(MRefEntity*, EventType) function is called with that entity as an argument. If the result of the cast is NULL, the Model::notify_observer function returns without doing anything further. During the construction of an MRefXxx object, CubitObserver::notify_static_observers gets called twice, once from the RefXxx::RefXxx constructor, and once from MRefXxx::MRefXxx. The first call does nothing, while the second call results in the entity being added to the global entity list.

Table 33: Events passed to CubitObserver::notify_static_observers(EventType) from various functions in CUBIT.

• Event	• Called from…	• Action
• VGI_BODY_CONSTRUCTED	• GeometryModifyTool:: make_Body	• Adds Body and its descendents to graphics display lists

• FREE_REF_ENTITY_CONSTRUCTED	• GeometryModifyTool::make_Xxx • (Xxx = Face, Edge, Vertex)	• Adds entity (Face, Edge, etc.) to graphics, then calls DrawingTool::make_topmost for that entity
• MODEL_ENTITY_CONSTRUCTED	• MRefXxx::MRefXxx, • RefXxx::RefXxx • (constructors)	• Adds entity to global entity list

5.5.3 Destruction

When an entity is destructed, its observers must be notified so that they can take the proper actions prior to that entity being removed. For example, when an entity which is contained in a group is destroyed, that entity must be removed from the group. Also, static observers should be notified of the entity being destroyed. Inside each MRefXxx::~MRefXxx function are calls to CubitObservable::remove_from_observers() and CubitObserver::notify_static_observers(this, MODEL_ENTITY_DESTRUCTED). The first function removes the observable from any observers that are observing it, while the second notifies static observers that the entity is about to be destroyed. The RefXxx::~RefXxx destructors do not need to call remove_from_observers(), since this function is called from CubitObservable::~CubitObservable(). This function must be called in the MRefXxx destructors because some observers need to know that the observable is an MRefEntity. If the remove_from_observers() function was not called in the leaf class destructors, by the time the function was called in the CubitObservable destructor the link between MRefEntity and RefEntity would be lost.

5.6 Attributes

There are three types of attributes implemented by CUBIT: mesh-specific attributes, entity names, and CUBIT owner attributes. The first two types of attributes are implemented as application-specific attributes using the techniques described in Section 4.3. CUBIT owner attributes are implemented outside the general attribute mechanism.

Documentation for the various mesh-specific attributes is found in a separate document. Entity names and CUBIT owner attributes are described below.

5.6.1 Entity Name

Although the Entity Name attribute is implemented using the general attributes capability in CGM, there are a few things that are done for entity names that diverge from a typical attribute. These differences are:

- Entity names are written back onto a solid model entity after being read from that entity, and after each change to the entity name on a RefEntity object. This is done to facilitate name persistence through booleans. This is implemented in RefEntityName::add_refentity_name, by calling CAEntityName::update after adding the entity name.
- In the case where default names are used, the entity names are not immediately written back to the solid model entities, since these names are not meant to persist across boolean operations.

5.6.2 CUBIT owner

The CUBIT owner attribute is used during the CUBIT session, to relate ACIS objects to their corresponding CGM objects. The data in the CUBIT owner attribute consists of a pointer to an AcisBridge object, from which a pointer to the corresponding TopologyEntity (and RefEntity) object can be found. Although the CUBIT owner attribute is stored in the ACIS .sat file, it is meaningless outside an active CUBIT session.

Besides its use when geometry is not changing (to complete the bi-directional link between TopologyBridge and ACIS objects), the CUBIT owner attribute is also used to implement persistent objects in CUBIT. In this mechanism, the CUBIT owner attribute pointers are used to determine which CUBIT entities correspond to ACIS entities which survive boolean operations, and which ones did not (and therefore should be deleted). Ultimately, the CUBIT owner attribute will be used to determine when the data in any other CUBIT attribute should be updated, for example to detect when the size of a geometric entity changes.

6 Summary and Conclusions

This report shows how to use CGM to construct geometry applications (User's Guide) and how to extend CGM to provide enhanced functionality on top of the geometry classes (Developer's Guide). CGM has been used in both ways in various applications at Sandia National Laboratories.

Several improvements have been made to CGM recently which are not described in this report. Those include distributing geometric models on parallel computers [1], porting CGM to various other solid modeling engines (including SolidWorks and Granite), and implementation of a common interface to geometry based on CGM. Please contact the author if additional information is needed about these items.

7 References

[1] ACIS 3D Toolkit, http://www.spatial.com/Products/Toolkit/toolkit.htm.

[2] S. J. Owen, D. R. White, T. J. Tautges, "Facet Based Surfaces for 3D Mesh Generation", Proc. 11th Int. Meshing Roundtable, Sandia National Laboratories, Albuquerque, New Mexico, September 2002.

[3] "SolidWorks API", SolidWorks Corp., http://www.solidworks.com/html/Products/api/.

[4] "Application Programming Toolkit", PTC, Inc, http://www.ptc.com/products/proe/app_toolkit.htm.

[5] Jason Kraftcheck, "Virtual Geometry: A Mechanism for Modification of CAD Model Topology For Improved Meshability", Master's Thesis, University of Wisconsin-Madison, December, 2000.

[6] T. D. Blacker et al., 'CUBIT mesh generation environment, Vol. 1: User's manual', SAND94-1100, Sandia National Laboratories, Albuquerque, New Mexico, May 1994, http://endo.sandia.gov/cubit/release/doc-public/Cubit_UG-4.0.pdf

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[8] Timothy J. Tautges, Hong-Jun Kim, "On the Partitioning and Initialization of Solid Geometry Models on Parallel Computers", 8th International Conference on Numerical Grid Generation in Computational Field Simulations, Honolulu, HA, June 2002.

Appendix A CGM Licensing

CGM will be released under an LGPL license; see http://cubit.sandia.gov/CGM for details. However, the LGPL-released version of CGM will not contain the ACIS modeling engine code, because of restrictions of the ACIS license. To obtain this code, institutions must themselves be licensed to use ACIS. Once this license is verified by Sandia, the CGM code calling ACIS functions can be obtained. Please contact cubit-dev@sandia.gov for details. CGM can be built and used without the ACIS functionality, for example using the facet-based modeling capability.

Appendix B CGM, CUBIT Class Diagrams

Query/Modify Tools, Engines

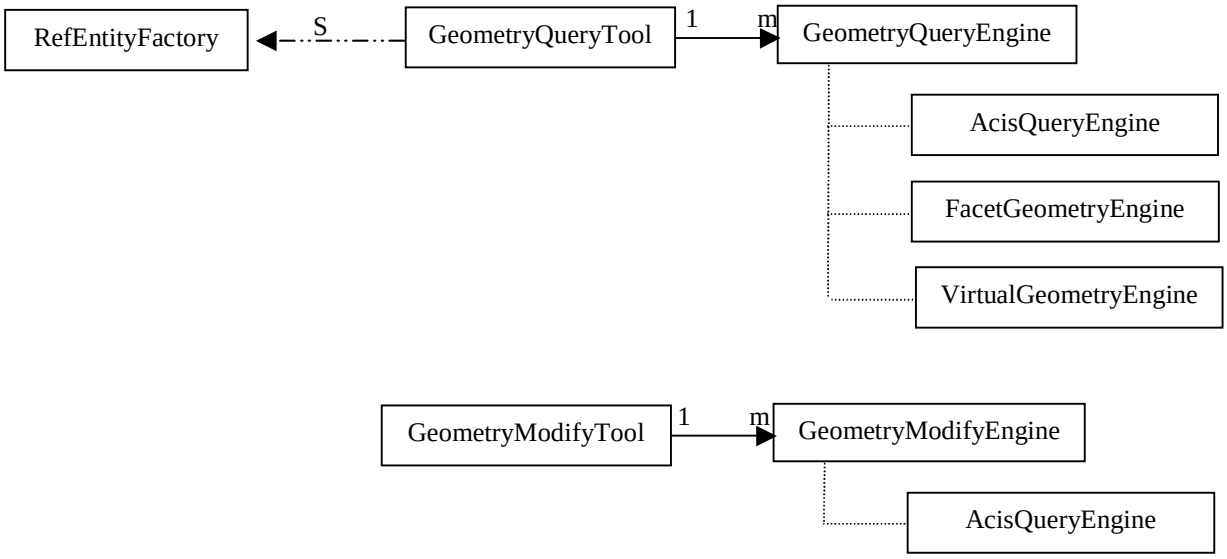

Figure 3: Class hierarchy for CGM geometry query and modify tools and engines. These classes are stored in CUBIT_BASE_DIR/geom subdirectory.

Topology Entities

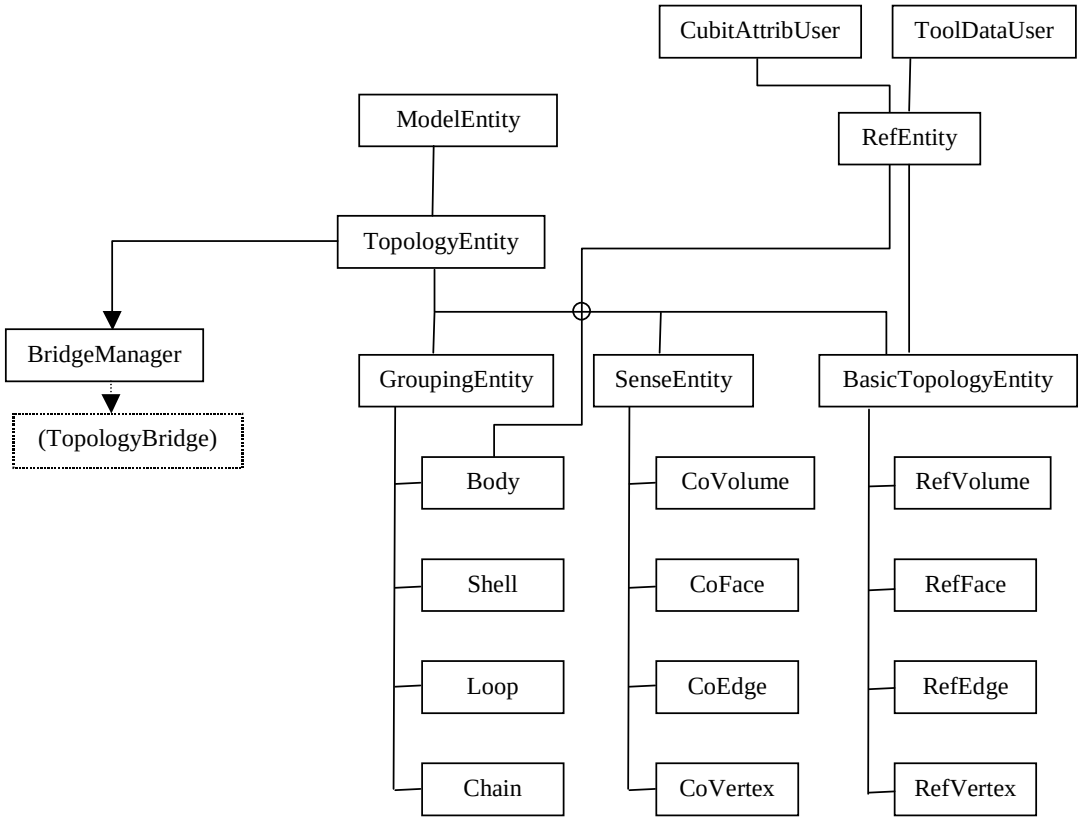

Figure 4: Class hierarchy for CGM topology entity classes. These classes are stored in CUBIT_BASE_DIR/geom subdirectory.

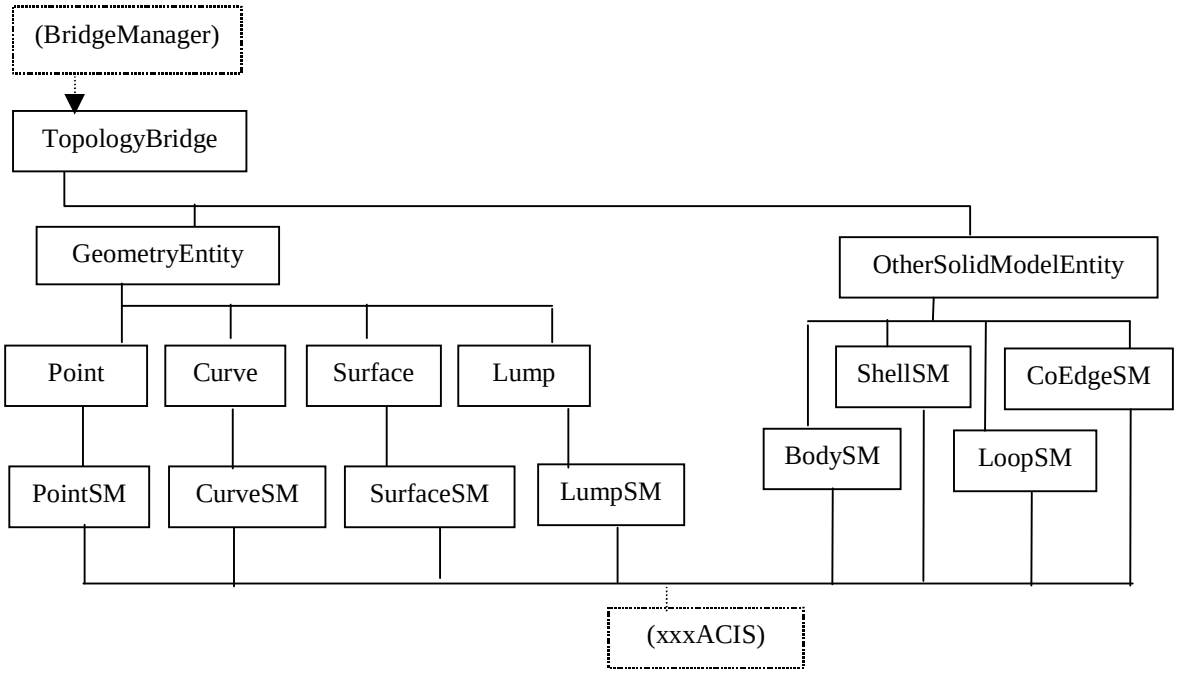

Figure 5: Class hierarchy for CGM geometry classes. These classes are stored in CUBIT_BASE_DIR/geom subdirectory.

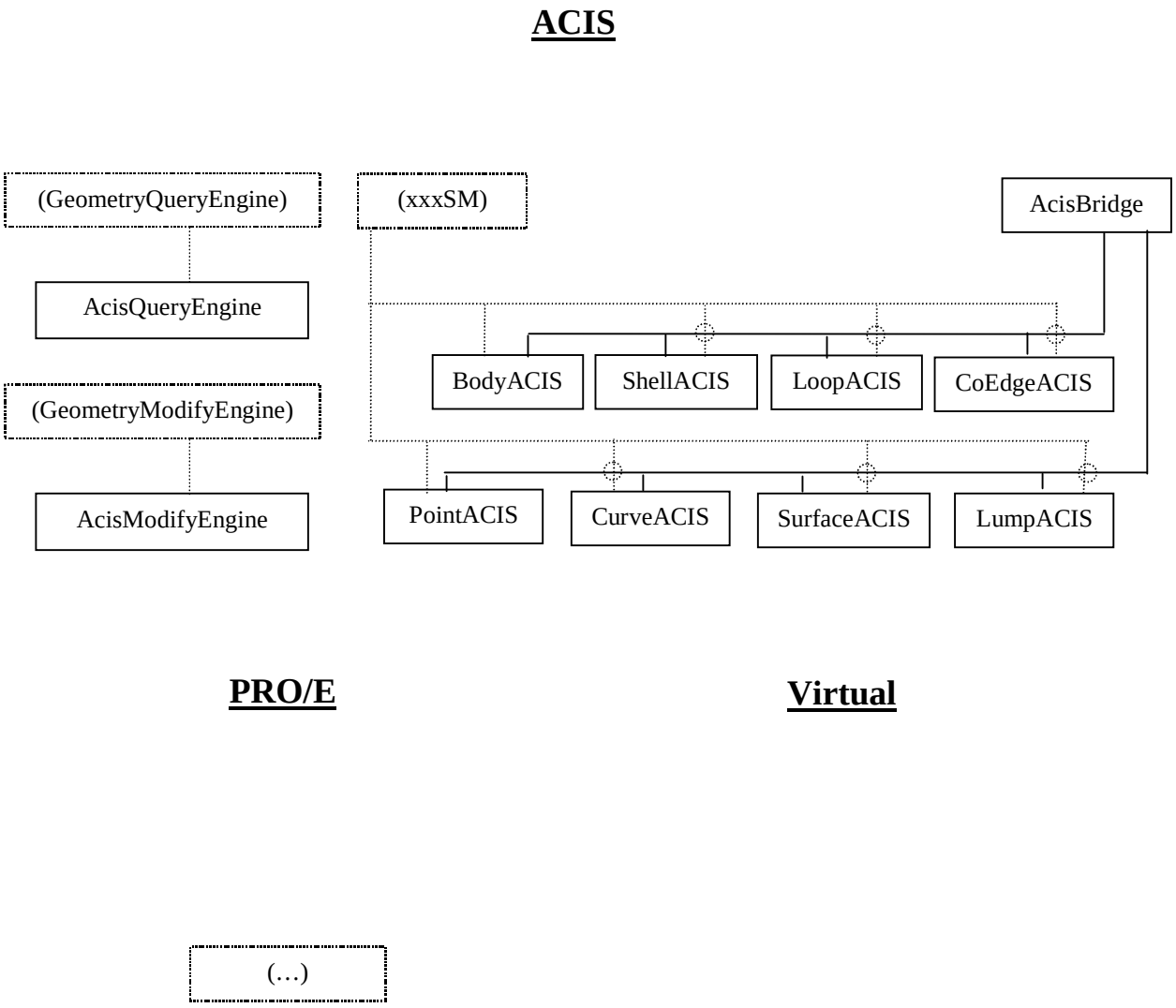

Figure 6: Class hierarchy for CGM geometry representation classes. These classes are stored in CUBIT_BASE_DIR/geom subdirectory.

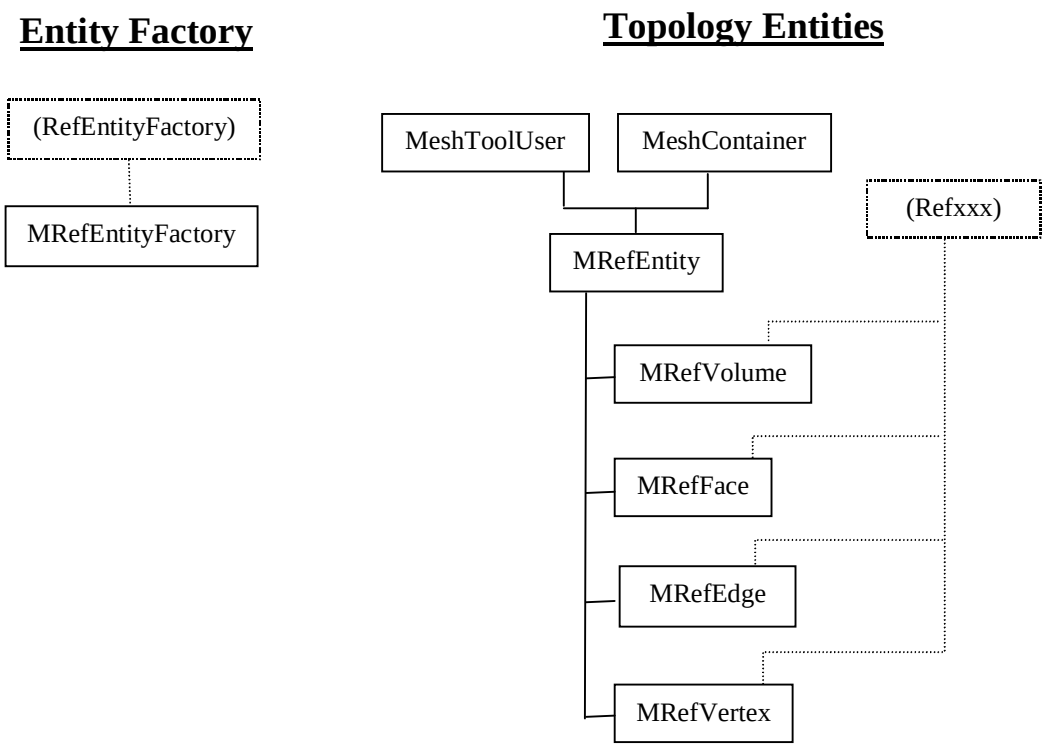

Figure 7: Class hierarchy for CUBIT factory and entity classes. These classes stored in CUBIT_BASE_DIR directory.

Distribution:

9226 (all members)
9220 Rob Leland
9114 Matt Hopkins
9143 Kevin Copps
9141 Steve Bova
15233 (I think) Arlo Ames
9231 Dave Hensinger
9143 Greg Sjaardema
6741 Len Lorence

Robert Haimes
Dept. of Aeronautics and Astronautics
Massachusetts Institute of Technology
77 Massachusetts Ave. 37-467
Cambridge, MA 02139-4307

Scientific Computation Research Center
Rensselaer Polytechnic Institute
110 8th Street
Troy, NY 12180-3590

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P.O. Box 808, L-661
Livermore, CA 94551

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P.O. Box 808, L-661
Livermore, CA 94551

Xiaolin Li
Applied Mathematics and Statistics
1-119 Mathematics Building
SUNY-Stony Brook
Stony Brook, NY 11794-3600

Anders Petersson
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Box 808, L-661
Livermore, CA 94551

Kyle Chand
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P.O. Box 808, L-661
Livermore, CA 94551

William Henshaw
Center for Applied Scientific Computing

Lawrence Livermore National Laboratory
Box 808, L-560
Livermore, CA 94551

Brian McCandless
Lawrence Livermore National Laboratory
Box 808
Livermore, CA 94551

David Hardin
Lawrence Livermore National Laboratory
Box 808
Livermore, CA 94551